

DIGITGRAPH
P. O. Box 4249
Tucson, Arizona 85717

PDP-11
FORTTRAN IV
COMPILER AND OBJECT TIME SYSTEM
PROGRAMMER'S MANUAL

SOFTWARE SUPPORT CATEGORY

The software described in this document is supported by DEC under Category I as defined on page iv of this document.

For additional copies, order No. DEC-11-LFIVA-A-D from Digital Equipment Corporation, Software Distribution Center, Maynard, Massachusetts 01754

First printing, August 1972

Your attention is invited to the last two pages of this document. The "How to Obtain Software Information" page tells you how to keep up-to-date with DEC's software. The Reader's Comments" page when filled in and mailed is beneficial to both you and DEC. All comments received are considered when documenting subsequent manuals.

Copyright © 1972 by Digital Equipment Corporation

This document is for information purposes and is subject to change without notice.

The following are registered trademarks of Digital Equipment Corporation:

Digital (logo)
DEC
DECtape
RSX-11

PDP-11
Comtex-11
RSTS-11
Unibus

PREFACE

FORTTRAN IV (FORMula TRANslation) is a problem-oriented language designed to permit scientists and engineers to express mathematical operations in a form with which they are familiar. The term FORTRAN is used interchangeably to designate both the FORTRAN IV language and the FORTRAN IV Compiler or translator. A program written in the FORTRAN language is called a FORTRAN source program.

A FORTRAN source program is composed of commands descriptive of the functions performed and computational elements expressed in a notation similar to that of mathematics. The source program is compiled by the FORTRAN IV Compiler into code which is then automatically assembled into a binary object module. The resultant object program is linked into a load module and run in conjunction with the FORTRAN Object Time System (OTS) Library described in Chapter 11 of this manual.

There is no one-to-one correspondence between a FORTRAN statement and a machine-language instruction. Some FORTRAN statements generate many machine instructions while others yield none.

SOFTWARE SUPPORT CATEGORIES

Digital Equipment Corporation (DEC) makes available four categories of software. These categories reflect the types of support a customer may expect from DEC for a specified software product. DEC reserves the right to change the category of a software product at any time. The four categories are as follows:

CATEGORY I

Software Products Supported at no Charge

This classification includes current versions of monitors, programming languages, and support programs provided by DEC. DEC will provide installation (when applicable), advisory, and remedial support at no charge. These services are limited to original purchasers of DEC computer systems who have the requisite DEC equipment and software products.

At the option of DEC, a software product may be recategorized from Category I to Category II for a particular customer if the software product has been modified by the customer or a third party.

CATEGORY II

Software Products that Receive Support for a Fee

This category includes prior versions of Category I programs and all other programs available from DEC for which support is given. Programming assistance (additional support), as available, will be provided on these DEC programs and non-DEC programs when used in conjunction with these DEC programs and equipment supplied by DEC.

CATEGORY III

Pre-Release Software

DEC may elect to release certain software products to customers in order to facilitate final testing and/or customer familiarization. In this event, DEC will limit the use of such pre-release software to internal, non-competitive applications. Category III software is only supported by DEC where this support is consistent with evaluation of the software product. While DEC will be grateful for the reporting of any criticism and suggestions pertaining to a pre-release, there exists no commitment to respond to these reports.

CATEGORY IV

Non-Supported Software

This category includes all programs for which no support is given

CONTENTS

	<u>Page</u>
CHAPTER 1	INTRODUCTION TO PDP-11 FORTRAN IV
1.1	PDP-11 FORTRAN IV 1-1
1.2	SOFTWARE AND HARDWARE ENVIRONMENT 1-2
1.3	HOW TO USE THIS MANUAL 1-2
1.4	FORTRAN STATEMENT STRUCTURE 1-3
1.4.1	Formatting a FORTRAN Line 1-3
1.4.2	Statement Number Field 1-5
1.4.3	Continuation Lines 1-6
1.4.4	Comment Lines 1-6
1.5	PROGRAM UNIT STRUCTURE 1-6
CHAPTER 2	FORTRAN STATEMENT COMPONENTS
2.1	FORTRAN CHARACTER SET 2-1
2.2	FORTRAN CONSTANTS 2-1
2.2.1	Integer Constants 2-2
2.2.2	Real Constants 2-3
2.2.3	Double-Precision Constants 2-3
2.2.4	Octal Constants 2-4
2.2.5	Hexadecimal Constants 2-4
2.2.6	Complex Constants 2-5
2.2.7	Logical Constants 2-6
2.2.8	Hollerith Constants 2-6
2.3	FORTRAN VARIABLES 2-7
2.3.1	Integer Variables 2-8
2.3.2	LOGICAL*1 Variables 2-8
2.3.3	Array Variables 2-8
2.4	FORTRAN EXPRESSIONS 2-9
2.4.1	Arithmetic Expressions 2-10
2.4.2	Use of Parentheses 2-11
2.4.3	Mixed Mode Arithmetic Expressions 2-11
2.4.4	Logical Expressions 2-13
CHAPTER 3	ASSIGNMENT STATEMENTS
3.1	ARITHMETIC ASSIGNMENT STATEMENT 3-1
3.2	LOGICAL ASSIGNMENT STATEMENT 3-2
3.3	ASSIGN STATEMENT 3-2

CONTENTS

	<u>Page</u>
CHAPTER 4	CONTROL STATEMENTS
4.1	GOTO STATEMENTS 4-1
4.1.1	Unconditional GOTO Statement 4-1
4.1.2	Computed GOTO Statement 4-1
4.1.3	Assigned GOTO Statements 4-2
4.2	IF STATEMENTS 4-3
4.2.1	Arithmetic IF Statement 4-3
4.2.2	Logical IF Statement 4-4
4.3	DO STATEMENT 4-4
4.4	CONTINUE STATEMENT 4-7
4.5	PAUSE STATEMENT 4-8
4.6	STOP STATEMENT 4-8
4.7	END STATEMENT 4-9
CHAPTER 5	SPECIFICATION STATEMENTS
5.1	IMPLICIT STATEMENT 5-1
5.2	TYPE DECLARATION STATEMENT 5-3
5.3	DIMENSION STATEMENT 5-5
5.3.1	Array Storage 5-6
5.4	COMMON STATEMENT 5-8
5.5	EQUIVALENCE STATEMENT 5-11
5.5.1	Equivalencing Array Variables 5-12
5.5.2	EQUIVALENCE and COMMON Interaction 5-13
5.5.3	EQUIVALENCE and BYTE Arrays 5-14
5.6	EXTERNAL STATEMENT 5-14
5.7	DATA STATEMENT 5-15
CHAPTER 6	FORTRAN FUNCTIONS AND SUBROUTINES
6.1	PROCEDURES 6-1
6.2	FORTRAN LIBRARY FUNCTIONS 6-2
6.3	ARITHMETIC STATEMENT FUNCTIONS 6-2
6.4	EXTERNAL SUBPROGRAMS 6-5
6.4.1	Subprogram Parameters 6-5
6.4.2	Adjustable Dimensions 6-8
6.4.3	FUNCTION Subprograms 6-8
6.4.4	SUBROUTINE SUBPROGRAMS 6-9
6.4.5	CALL Statement 6-12
6.4.6	RETURN Statement 6-12
6.4.7	Assembly Language Subprograms 6-13

CONTENTS

	<u>Page</u>
6.4.8 BLOCK DATA Subprogram	6-15
6.5 SYSTEM SUBROUTINES	6-15
6.5.1 PDUMP	6-16
6.5.2 SETERR	6-18
6.5.3 SETFIL	6-19
6.5.4 RANDU, RAN	6-21
6.5.5 EXIT	6-22
6.5.6 DATE	6-22
6.5.7 TIME	6-23
6.5.8 SSWTCH	6-25
6.5.9 TSTERR	6-25
6.5.10 Character String Arguments to System Subroutines	6-26
 CHAPTER 7 PROGRAM OVERLAY FACILITY	
7.1 INTRODUCTION TO THE OVERLAY FACILITY	7-1
7.2 COMMUNICATION AMONG RESIDENT AND OVERLAY ROUTINES	7-2
7.3 FORMAT CONVERSIONS AND I/O ROUTINES	7-2
7.4 LINK, THE RUN-TIME OVERLAY SUPERVISOR	7-4
7.5 CALLING AN OVERLAY FILE	7-6
7.5.1 Overlay Transfer Paths	7-7
7.5.2 Search for Overlay Files	7-8
7.6 OPERATING PROCEDURES	7-9
7.6.1 Linker Command String Formats	7-9
7.6.2 Operations Performed by the Linker	7-10
7.7 ERROR PROCEDURES AND MESSAGES	7-11
7.8 ASSEMBLY LANGUAGE OVERLAYS	7-13
7.8.1 Global Declarations	7-13
7.8.2 Stack Usage	7-13
7.8.3 Register Usage	7-14
7.8.4 COMMON Communication	7-14
7.9 PROGRAMMING NOTES	7-16
7.10 A SUMMARY OF THE MAJOR FEATURES OF THE OVERLAY FACILITY	7-17
7.11 PROGRAMMING EXAMPLE OF OVERLAY SYSTEM CREATION	7-17

CONTENTS

	<u>Page</u>
7.12 CALL RUN SUBROUTINE	7-32
 CHAPTER 8 FORMAT STATEMENTS	
8.1 FORMAT STATEMENT	8-1
8.2 I FORMAT CONVERSIONS	8-3
8.3 O FORMAT CONVERSIONS	8-4
8.4 F FORMAT CONVERSIONS	8-4
8.5 E FORMAT CONVERSIONS	8-5
8.6 D FORMAT CONVERSIONS	8-6
8.7 G FORMAT CONVERSIONS	8-8
8.8 COMPLEX I/O	8-9
8.9 SCALE FACTORS	8-9
8.10 L FORMAT CONVERSIONS	8-10
8.11 A FORMAT CONVERSIONS	8-11
8.12 ALPHANUMERIC DATA WITHIN FORMAT SPECIFICATION	8-12
8.13 RECORD LAYOUT SPECIFICATIONS	8-13
8.14 CARRIAGE CONTROL	8-14
8.15 REPETITION OF FIELDS AND GROUPS AND MULTIPLE RECORDS	8-15
8.16 FORMATS STORED AS DATA	8-16
 CHAPTER 9 I/O AND DEVICE HANDLING STATEMENTS	
9.1 INTRODUCTION TO FORTRAN I/O	9-1
9.1.1 I/O Devices	9-3
9.1.2 I/O Records	9-3
9.1.3 I/O Lists	9-4
9.2 FORMATTED I/O	9-6
9.2.1 Formatted READ Statement	9-6
9.2.2 Formatted WRITE Statement	9-7
9.2.3 PRINT Statement	9-8
9.3 UNFORMATTED I/O	9-9
9.3.1 Unformatted READ Statement	9-9
9.3.2 Unformatted WRITE Statement	9-10
9.4 DIRECT ACCESS I/O	9-10
9.4.1 DEFINE FILE Statement	9-10

CONTENTS

	<u>Page</u>
9.4.2 FIND Statement	9-12
9.4.3 Direct Access READ Statement	9-12
9.4.4 Direct Access WRITE Statement	9-13
9.5 TRANSFER OF CONTROL ON ERROR CONDITION	9-13
9.6 DEVICE CONTROL STATEMENTS	9-14
9.6.1 BACKSPACE Statement	9-14
9.6.2 REWIND Statement	9-14
9.6.3 END FILE Statement	9-15
9.7 ENCODE AND DECODE	9-15
 CHAPTER 10 FORTRAN/DOS I/O CONSIDERATIONS	
10.1 INPUT/OUTPUT OVERVIEW	10-1
10.2 FILE STRUCTURES AND FORMATS	10-3
10.2.1 FORTRAN Formatted I/O	10-5
10.2.2 FORTRAN Unformatted I/O	10-6
10.2.3 FORTRAN Direct-Access I/O	10-7
10.3 FORTRAN I/O ERROR HANDLING	10-8
10.4 FORTRAN DEVICE ASSIGNMENTS	10-9
10.5 THE FORTRAN DEVICE TABLE, \$DEVTB	10-11
10.6 I/O EXAMPLE	10-13
 CHAPTER 11 FORTRAN OPERATING ENVIRONMENT	
11.1 FORTRAN OBJECT TIME SYSTEM	11-1
11.2 THREADED CODE	11-1
11.2.1 Entry to Polish Mode	11-3
11.2.2 Exit from Polish Mode	11-4
11.3 FORTRAN RUN-TIME MEMORY ORGANIZATION	11-5
11.4 DEBUGGING FORTRAN PROGRAMS	11-7
11.5 FORTRAN OTS ERROR PROCESSING	11-9
 CHAPTER 12 OPERATING PROCEDURES	
12.1 USING THE FORTRAN SYSTEM	12-1
12.1.1 Filename Specifications	12-1
12.1.2 Compilation and Linking Procedures	12-3
12.1.3 Compile-Load-and-Go Operation	12-5

CONTENTS

	<u>Page</u>
12.1.4 FORTRAN Library Usage	12-6
12.2 SWITCH OPTIONS	12-6
12.3 FORTRAN OUTPUT LISTING FORMAT	12-8
12.4 COMPILE-TIME MEMORY REQUIREMENTS	12-10
APPENDIX A FORTRAN CHARACTER SET	A-1
APPENDIX B FORTRAN WORD FORMATS	B-1
APPENDIX C FORTRAN LANGUAGE SUMMARY	C-1
APPENDIX D FORTRAN LIBRARY FUNCTIONS	D-1
APPENDIX E FORTRAN ERROR DIAGNOSTICS	E-1
APPENDIX F FORTRAN DEVICE TABLE LISTING (\$DEVTB)	F-1

CHAPTER 1

INTRODUCTION TO PDP-11 FORTRAN IV

1.1 PDP-11 FORTRAN IV

PDP-11 FORTRAN conforms to the American National Standard Institute (ANSI) FORTRAN IV specifications. It is also highly compatible with IBM 1130 FORTRAN. The following additions to ANSI FORTRAN IV are included in PDP-11 FORTRAN IV:

- a. Array subscripts may consist of any mixed-mode expression. If the result of the expression is not an integer, it is truncated and converted to integer format before being used as a subscript.
- b. Hexadecimal and octal constants are allowed within DATA statements.
- c. Hollerith constants can be delimited by single quote (apostrophe) characters.
- d. General mixed mode expressions are allowed for all data types including complex.
- e. Optional label list in assigned GOTO.
- f. The following statements have been added to handle random access I/O:

```
DEFINE FILE
FIND
READ (a'b)
WRITE (a'b)
```

- g. The IMPLICIT statement is provided to allow re-definition of the default variable typing rules (normally I-N indicate integer variables, and A-H and O-Z indicate real variables).
- h. The data types
LOGICAL*1
INTEGER*2
REAL*4
REAL*8
are provided.
- i. An O (octal) field specification and T (tab) field specification are allowed with format descriptions.
- j. The I/O conditions END=n and ERR=n can be specified on any READ or WRITE statement to transfer control upon detection of an end-of-file or error condition.

1.2 SOFTWARE AND HARDWARE ENVIRONMENT

PDP-11 FORTRAN is used with the PDP-11 DOS (Disk Operating System) Monitor (known as "the Monitor" in this document), PAL-11R or MACRO-11, Link-11, Libr-11 (Librarian), and Edit-11 (Editor). These system programs are available from the DEC Software Distribution Center and are delivered with each hardware configuration capable of supporting them.

The FORTRAN IV Compiler outputs code suitable for assembly by the PAL-11R or MACRO-11 Assemblers. The user has the option of having his FORTRAN program compiled and assembled in one step (which is normally done) or compiled into assembly language for later assembly (indicated by means of a switch option to the Compiler). FORTRAN programs may call assembly language subroutines. The FORTRAN Library routines are callable from either FORTRAN or assembly language code.

PDP-11 FORTRAN programs can be compiled and run on hardware configurations which support the PDP-11 DOS Monitor and have 12K or more of core. The minimum hardware requirements are: 12K of core; RF11, RK11, or RC11 disk; high-speed reader/punch and/or DECTape; and an ASR-33 Teletype¹ terminal. PDP-11 FORTRAN IV supports all standard hardware options supported by the DOS Monitor.

1.3 HOW TO USE THIS MANUAL

This manual is primarily a reference document describing the PDP-11 FORTRAN IV language. It is not intended to teach the reader how to program in FORTRAN. Numerous texts on this subject are available on the market for the novice programmer. The bulk of this manual describes details of the language statements and descriptions of the file handling, library, internal structure, and Object Time System. Appendices provide concise summaries of material in tabular form.

¹ Teletype is a registered trademark of the Teletype Corporation.

Although designed as a reference tool, this manual contains a number of programming examples to show the use of statements in context. Internal details supplied will aid the more advanced FORTRAN programmer.

1.4 FORTRAN STATEMENT STRUCTURE

Statements are the basic functional units of the FORTRAN language. An executable statement performs a calculation, does I/O, or directs control of the program; a nonexecutable statement provides the Compiler with information regarding variable structure, array allocation, storage requirements, etc.

All categories of FORTRAN statements are coded in a standard format for ease of program preparation. Figure 1-1 shows a FORTRAN coding form (there are many types available) with five miscellaneous lines written on it. FORTRAN coding forms lend themselves to paper tape input as well as punched card input preparation. The form is divided into four major sections:

- a. Statement number field - columns 1-5
- b. Line continuation field- column 6
- c. Statement field- - - - columns 7-72
- d. Sequence field, ignored
by the FORTRAN Compiler- columns 73-80

1.4.1 Formatting a FORTRAN Line

Punched card input for the PDP-11 FORTRAN Compiler is formatted as shown on the coding form in Figure 1-1. This is the standard format for FORTRAN punched cards on most computer systems.

When programs are prepared on-line with Edit-11, statements can be typed in the same manner as described for cards or can use the TAB character as follows:

- a. A TAB causes the character counter to be set to 6 or 7 depending on whether or not the character following the TAB is numeric. If the character is not numeric, it is assumed to be the start of a statement proper. If the character following the tab is numeric, the line is assumed to be a continuation line.
- b. A one to five digit statement label may precede the TAB character.

In order to obtain the formatting shown on the coding form in Figure 1-1, the user would type the following to Edit-11:

123→X=12.+3.	or	123 <del> </del> <del> </del> X=12.+3.
→A=B+C	or	<del> </del> <del> </del> <del> </del> A=B+C
→1A1'ABC <del> </del> '	or	<del> </del> <del> </del> <del> </del> 1A1'ABC <del> </del> '
C→COMMENT <del> </del> LINE	or	C <del> </del> <del> </del> <del> </del> COMMENT <del> </del> LINE
CCOMMENT <del> </del> LINE	or	CCOMMENT <del> </del> LINE

where:

- - indicates typing of the TAB character (CTRL/TAB)
- indicates typing of the SPACE bar.

As specified in Section 2.1, the TAB character is not included in the legal FORTRAN character set; therefore, the TAB character can be used only at the beginning of a line to indicate spacing to the continuation or statement field. FORTRAN ignores spaces within each field (except within Hollerith constants). Spaces can generally be inserted at the user's convenience to increase program legibility.

1.4.2 Statement Number Field

A-statement number consists of one to five digits in columns 1 to 5. Leading zeros and all blanks in this field are ignored. Any statement referenced by another statement must have a statement number.

Statement numbers must be unique. If two statements within a program unit have the same statement number, the error is noted at assembly time.

1.4.3 Continuation Lines

Any FORTRAN statement can be continued onto more than one physical source line. A non-zero or non-blank character is placed in column 6 of all continuation lines (or cards). When using Edit-11, the TAB character followed by a non-zero numeric character is used to indicate a continuation line.

Up to five continuation cards or lines are normally allowed by PDP-11 FORTRAN IV; however, this number can be changed to any number in the range 0 to 99 through use of the /CO switch (see Section 12.2).

1.4.4 Comment Lines

A comment can be inserted into a FORTRAN program by making the first character of the line a C. If the C character is detected in column 1 of a card or as the first character after the preceding carriage return/line feed, all information on that line, up to the next carriage return/line feed, is disregarded. A comment line must not precede a continuation line. Continuation lines cannot be used to continue comments. All comment lines are preceded by the letter C.

1.5 PROGRAM UNIT STRUCTURE

A FORTRAN program unit is a FORTRAN main program or a subprogram. Other than BLOCK DATA subprograms, which are a special class (described in Section 6.4.8), program units are constructed of FORTRAN statements in the following order:

- a. Subprogram statement, if the program unit is a subprogram.
- b. IMPLICIT statement, if used.
- c. Specification statements (other than IMPLICIT). Those specification statements determining the storage required by a variable or array precede statements which initialize the values of the variable or array. DATA statements must follow all other specification statements.
- d. Statement function definitions, if used.
- e. Executable program statements. Each program unit (except BLOCK DATA subprograms) must have at least one executable program statement.
- f. END statement.

CHAPTER 2

FORTRAN STATEMENT COMPONENTS

2.1 FORTRAN CHARACTER SET

The character set from which FORTRAN statements can be constructed is as follows:

the letters A-Z	
the digits 0-9	
blank	/ slash
= equal sign	(left parenthesis
+ plus sign	) right parenthesis
- minus sign	, comma
* asterisk	. decimal point
\$ dollar sign.	

The TAB character can be used only at the beginning of a line to indicate spacing to the continuation or statement field. Other printing characters can appear within a FORTRAN program only as part of a comment line or a Hollerith constant (text string).

While the PDP-11 DOS Monitor provides device independence for input to the FORTRAN Compiler, it is required that FORTRAN source programs be in ASCII code. Programs intended for compilation by the PDP-11 FORTRAN IV Compiler and prepared on cards, paper tape, or magtape from other Compiler systems must conform to this assumption. (See the PDP-11 DOS device handler documentation for external codes.)

2.2 FORTRAN CONSTANTS

A constant is a self-defining value which does not change from one execution of the program to the next. In general, constants can be numeric, logical, or character string entities.

Types of constants allowed in PDP-11 FORTRAN IV include:

integer	}	numeric constants
real		
double precision		
octal		
hexadecimal		
complex		
logical	-	logical constants
Hollerith	-	character string constants

Storage formats for each type of constant are described in Appendix B.

2.2.1 Integer Constants

An integer constant is a string of numeric characters from one to five digits long and is interpreted as a decimal integer. Integers are 16-bit signed quantities. A negative integer is preceded by a minus sign; a positive integer is optionally preceded by a plus sign.

An integer constant must lie within the range -32768 to +32767 (which is -2^{15} to $2^{15} - 1$). An integer constant normally occupies two words of PDP-11 storage; one word contains the integer value and one word of padding is supplied (see Appendix B). This allows an integer to be equivalenced to a real number as defined by the ANSI FORTRAN specification. (See Section 12.2 for a way to select single word integers.)

The result of any operation which exceeds 16 bits results in an overflow diagnostic by the Object Time System.

Examples of integers acceptable to PDP-11 FORTRAN follow:

```
3564
+14
-26357
```

Commas and decimal points are not allowed in integer constants.

2.2.2 Real Constants

A real (or floating-point) constant is a string of numeric characters including a decimal point. A real constant can contain any number of digits, but only the leftmost eight digits are significant. The plus sign can be omitted for positive values.

A real constant can alternately be expressed in scientific notation with a decimal exponent, represented by the letter E, and a signed integer constant. The decimal point with the real number is optional if there is an exponent. The exponent field cannot be blank, but can be zero.

A real constant has 24 bits of precision, or somewhat more than seven decimal digits. The magnitude must lie within the range $.14 \times 10^{-38}$ to 1.7×10^{38} . Real constants occupy two words of PDP-11 storage (see Appendix B).

Examples of real constants follow:

15.	
.579	
0.0	
-10.685	
5.0E3	(i.e., 5000.)
4.2E+3	(i.e., 4200.)
2.0E-3	(i.e., .002)
5E0	(i.e., 5.0)

2.2.3 Double-Precision Constants

A double-precision constant is a string of numeric characters including a decimal point and the letter D followed by an integer exponent. The decimal point is optional. A double-precision constant can contain any number of digits, but only the leftmost 16 digits are significant. A double-precision constant is always expressed in scientific notation with a decimal exponent (represented by the letter D, and a signed integer constant). The exponent field cannot be blank, but may be zero. The plus sign preceding the constant and/or preceding the exponent can be omitted for positive values.

A double-precision constant has 56 bits of precision or about 16 decimal digits. The magnitude of a double-precision constant must lie between $.14 * 10^{-38}$ and $1.7 * 10^{38}$. Double-precision constants occupy four words of PDP-11 storage. Examples of double-precision constants are shown below:

24.6713264837465DØ	
2.5D2	(i.e., 25Ø. to 16 digits precision)
3.6D-2	(i.e., .Ø36 to 16 digits precision)
5.6DØ	(i.e., 5.6 to 16 digits precision)

2.2.4 Octal Constants

The general form for specifying an octal constant is to precede a string of one to six octal digits (Ø to 7 inclusive) with the " character. For example:

```
"12Ø
"Ø
"177777
```

Octal constants of this form can appear anywhere an integer constant is valid. An octal constant can also be a string of one to six octal digits preceded by the letter O. Octal constants of this form are valid only in DATA, STOP and PAUSE statements. Examples are shown below:

```
O12Ø
OØ
O177777
```

The maximum value of an octal constant is 177777.

2.2.5 Hexadecimal Constants

A hexadecimal constant is a string of one to four hexadecimal digits (Ø to 9, and A to F, inclusive) preceded by the letter Z. The hexadecimal digits are as follows:

<u>hexadecimal digit</u>	<u>decimal equivalent</u>
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
A	10
B	11
C	12
D	13
E	14
F	15

A hexadecimal constant is valid only in a DATA statement.
The maximum value of a hexadecimal constant is FFFF. Examples:

ZA34
ZC1
ZFFFF

2.2.6 Complex Constants

A complex constant is written as an ordered pair of real constants separated by a comma and enclosed in parentheses. The first constant represents the real part of the complex number and the second constant represents the imaginary part; each may be signed. The parentheses are part of the constant proper. A complex constant is represented internally by four consecutive words of PDP-11 storage. Examples follow:

(.70712,-1.70712)
(8.763E3,2.297)

2.2.7 Logical Constants

The two logical constants `.TRUE.` and `.FALSE.` are represented internally by the integer values `-1` and `0`, respectively. (The delimiting periods are part of the constant and always appear). Any non-zero value is considered true when tested by a logical IF statement; `-1` is equivalent to the octal number `177777`. These values can be entered via DATA or input statements as `.TRUE.` and `.FALSE.` (see Sections 5.7 and 8.10). Both arithmetic and logical operators can be used with logical constants which are treated as one-word integer values.

2.2.8 Hollerith Constants

A Hollerith (or literal) constant is a string of up to 255 characters which can be represented in one of two ways (where `Ø` indicates a blank):

- a. An integer constant indicating the number of characters, followed by an H, followed by a character string:

`5HWORDS`

`4H1Ø23`

- b. A character string enclosed in single quote characters:

`'WORDS'`

`'1Ø23'`

The single quote character can be included in the character string if it is immediately preceded by another single quote character. For example:

`'DON''T'`

Hollerith constants can be entered in DATA or input statements as a string of one or two ASCII characters per integer variable, one to four characters per real variable, and one to eight characters per complex or double-precision variable. Hollerith constants are stored in memory as byte strings, filling up to word boundaries. If a Hollerith constant has an odd number of characters, a blank is appended to the end of the constant.

A Hollerith constant is treated as an integer quantity when it appears in an arithmetic context. For example:

I = 'ABC' +34

is equivalent to

I = 'AB' +34

The third and any subsequent characters are ignored. The integer value of 'AB' can be determined by examining the storage format, which is

15	Ø
ASCII value of B	ASCII value of A

2.3 FORTRAN VARIABLES

A simple FORTRAN variable is a named quantity whose value may change during execution of a program. A variable name is a string of one to six alphanumeric characters, the first of which must be alphabetic. Variable names longer than six characters are rejected by the FORTRAN Compiler, and an error diagnostic is printed.

<u>Legal Names</u>	<u>Illegal Names</u>
ALPHA	2A (begins with a number)
MAX	MAXIMUM (more than 6 characters)
A34	

A variable has a principal attribute type, just as constants are of one type or another. The type of the variable indicates the type of value it can be assigned, its precision, and its storage requirements.

Type INTEGER, REAL, LOGICAL, DOUBLE PRECISION or COMPLEX is assigned by means of an explicit type declaration statement (see Section 5.2), an IMPLICIT statement (see Section 5.1), or by virtue of the initial letter of the variable name.

Assuming no type declarations or IMPLICIT statements, if a variable name begins with I, J, K, L, M, or N, the variable is assumed to be an integer variable. All other initial letters indicate a real variable.

<u>Integer Variables</u>	<u>Real Variables</u>
MAX	REAL
IDAT	DATA1
NJOB	X

2.3.1 Integer Variables

Integer variables undergo arithmetic calculations with automatic truncation of any fractional part. For example, if the current value of K is 5 and the current value of J is 9, J/K yields 1 as a result. Integer values can be converted to real values by the FLOAT function or by an arithmetic assignment statement. Integer arithmetic operations do check for overflow and print an error message if the result of an integer operation overflows 16 bits.

2.3.2 LOGICAL*1 Variables

Data items in LOGICAL*1 format are 8 bits long (1 byte), and are always accessed by means of a variable name defined as BYTE or LOGICAL*1 type. The range of numbers from +127 to -128 can be represented. Any arithmetic operation is performed internally by taking two 8-bit operands, extending the sign to a full word, and performing the desired operation. No diagnostics are issued for byte overflow; the result is truncated to an 8-bit quantity.

Logical and masking operations are performed with one byte of data at a time. LOGICAL*1 (or BYTE) array elements are stored in adjacent bytes.

2.3.3 Array Variables

The extent of a variable refers to the range of values which are referenced by a single variable name. A scalar variable represents a single quantity; i.e., BETA represents a single numeric value.

An array variable represents a one, two, or three dimensional array of quantities. A given variable name cannot be used as both a simple variable and an array name in the same program unit. An array element is denoted by the array name followed by a subscript list enclosed in parentheses. The subscript list can be any one, two, or three arithmetic expression(s) separated by commas.

An array can be defined with a DIMENSION or COMMON statement or as part of a type declaration statement. Array dimensioning is discussed in Section 5.3.

Any legal arithmetic expression can be used as a subscript of an array element once the array has been defined. Floating-point subscripts are converted to integers by truncation of the non-integer part before use.

The extent of an array variable is determined by the number of dimensions it is assigned. For example, the following are legal references to array elements:

One Dimensional Array:	A(1)
Two Dimensional Array:	REL(24,3)
Three Dimensional Array:	TR4(X*2,5,2*X)

Arrays are stored in increasing storage locations with the first subscript varying most rapidly. See Section 5.3.1 for a detailed description of array storage. For instance, the two-dimensional array B(J,K) is stored in the following order:

```
B(1,1),B(2,1),...,B(J,1)
B(1,2),B(2,2),...,B(J,2)
  ⋮
B(1,K),  ...    ,B(J,K)
```

References to an array variable must contain the number of subscripts with which the array was defined, the exception being those statements which can reference an entire array given only the array name.

2.4 FORTRAN EXPRESSIONS

FORTRAN expressions can be arithmetic, relational, or logical. Relational and logical expressions have logical values which can

be treated as integers. All expressions are combinations of elements and operators. The different types of expressions are discussed below.

2.4.1 Arithmetic Expressions

An arithmetic expression is a combination of arithmetic operators and elements. The arithmetic operators are as follows, presented in descending order of priority:

<u>Operator</u>	<u>Meaning</u>
**	exponentiation ($A^{**}2$)
-	unary minus (-2)
*,/	multiplication ($A*B$), division ($1/2$)
+,-	addition ($A+B$), subtraction ($D-4$)

An arithmetic expression may not contain adjacent arithmetic operators. In cases where two operations within an expression have the same priority (*,/ and +,-) the operations are performed from left to right. For example:

$-A^{**}2+B*C-1$

is performed as:

$(-(A^{**}2)) + (B*C) -1$

Additional computations (such as sine, cosine, square root) can be specified via library function references. A function reference acts as a basic element in an expression since all functions return a single value. (For example, $SQRT(4.0)$ returns the value 2.0 in an expression).

An arithmetic expression may consist of a single element:

B4
22.7
NEX(3)

or a combination of arithmetic operators and basic elements:

$-X+3$
 $A/2+DAT$
 $TAN(PI*M)$
 $B^{**}(-2)$

2.4.2 Use of Parentheses

Any arithmetic expression can be enclosed in parentheses and considered a basic element to be evaluated before use in the remainder of the expression. Parentheses can be used to change the order of evaluation. For example:

$$A*(B+C**2)$$

is evaluated as:

$$(A* (B+(C**2))$$

omitting the parentheses:

$$A*B+C**2$$

is evaluated as:

$$(A*B) + (C**2)$$

Within parentheses the order of operator evaluation is as follows:

- a. function evaluations
- b. exponentiation
- c. unary minus
- d. multiplication, division
- e. addition, subtraction

Parentheses cannot be used to imply multiplication. That is:

$$A(2)$$

is the second element of the array A, whereas

$$A*2$$

indicates twice the value of the variable A.

2.4.3 Mixed Mode Arithmetic Expressions

With few exceptions, any type of constant, variable or function can be combined with any other type in an arithmetic expression. The type of the resultant expression when any two types are combined is found in Table 2-1.

Table 2-1
Mixed Mode Arithmetic Results

	Byte	Integer	Logical	Real	Double Precision	Complex
Byte	Byte	Integer	Logical	Real	Double Precision	Complex
Integer	Integer	Integer	Integer	Real	Double Precision	Complex
Logical	Logical	Integer	Logical	Real	Double Precision	Complex
Real	Real	Real	Real	Real	Double Precision	Complex
Double Precision	Double Precision	Double Precision	Double Precision	Double Precision	Double Precision	Complex
Complex	Complex	Complex	Complex	Complex	Complex	Complex
The general rule for mixed modes is that the result has the type of the most significant operand.						
The following special rules apply for determining the type resulting from expressions of the form $A**B$: 1. If B is integer, the expression is of the same type as A; 2. If A and B are both real, the expression is real; 3. If A or B are double-precision, the expression is double-precision; 4. No other possibilities for A and B are legal.						
When a real number is converted to double-precision format, repeating decimals are not carried through the additional precision.						

Logical, octal, and Hollerith constants are treated as integer constants when combined with other elements in arithmetic expressions. The result of such a combination is then changed to the appropriate resultant mode.

Arithmetic operations can be performed between complex and other data types. Real, integer and double-precision values (represented by V) are converted, if necessary, to the complex value (V,0.0) prior to the arithmetic operations; V is a real value in this representation.

The only illegal mixed mode operations are :

B**X
I**R
I**D
C**R
C**D
X**B
X**C

where: B is a byte value
 I is an integer value
 R is a real value
 C is a complex value
 D is a double-precision value
 X is a value of any data type

2.4.4 Logical Expressions

A logical expression combines constants, logical variables, logical function references and arithmetic expressions using relational or logical operators. Logical masks can be represented by using octal constants. The logical constants .TRUE. and .FALSE. can be used in any logical expression and have the values -1 and 0, respectively. The result of a logical expression is the logical value true or false and uses one word of storage space. (Any non-zero integer value is considered true when tested by a logical IF statement).

A relational expression consists of two arithmetic expressions separated by a relational operator. The arithmetic expressions can be of any complexity and any combination of integer, real, or double-precision types. The relational operators are as follows (where A and B are arithmetic expressions):

<u>Relational Operator</u>	<u>Example</u>	<u>Meaning</u>
.GT.	A.GT.B	A is greater than B
.GE.	A.GE.B	A is greater than or equal to B
.LT.	A.LT.B	A is less than B
.LE.	A.LE.B	A is less than or equal to B
.EQ.	A.EQ.B	A is equal to B
.NE.	A.NE.B	A is not equal to B

Relational expressions can be written comparing values of different precision (having different data types). The value having the least precision is converted to the highest precision format. The following should be noted:

- a. Any real, integer, or double-precision value (represented by V) is converted to the complex value (V,Ø.Ø). V is a real value in this representation.
- b. When a real number is converted to double-precision format, repeating decimals are not carried through the additional precision. This, for instance, causes a real Ø.333... to be less than a double-precision Ø.333...

Only the relational operators .EQ. and .NE. can be used with complex expressions (complex quantities are equal if their corresponding parts are equal).

The value of a relational expression is always either true or false (-1 or Ø, no other value). The following are relational expressions:

```
A.GT.B
A**2+C.LT.NET*B+4
AB/C.LE.AB/D
```

Parentheses can be used in relational expressions. For example:

```
(A+B)*C.EQ.(D+1)**2
(A(1)+1).GT.(B(1))
```

A logical expression may also contain logical operators which operate on relational expressions. The logical operators are as follows:

<u>Logical Operator</u>	<u>Example</u>	<u>Meaning</u>
.NOT.	.NOT.A	has the value .TRUE. only if A is .FALSE. and has the value .FALSE. only if A is .TRUE.
.AND.	A.AND.B	has the value .TRUE. only if A and B are both .TRUE., and has the value .FALSE. if either A or B is .FALSE.
.OR.	A.OR.B	(inclusive OR) has the value .TRUE. if either A or B or both are .TRUE. and is .FALSE. only if both A and B are .FALSE.

A logical expression can consist of basic logical elements or a combination of logical elements. For example, the following are all logical expressions (relational expressions are a subtype of logical expression):

```
.TRUE.
X.GE.3.14159
TV(1).AND.INDEX
B(M).OR.(K.EQ.LIMIT)
```

No two relational or logical operators can appear in sequence, except in the case where .NOT. appears as the second of two logical operators. Any logical expression can be preceded by the unary operator .NOT. as follows:

```
.NOT.T
.NOT.(X+7.GT.Y+Z)
B(M).AND..NOT.TV.OR.R    which is the same as:
                           (B(M).AND.(.NOT.TV)).OR.R
```

However, where .NOT. appears next to an arithmetic operator, the arithmetic expression should appear in parentheses. For example:

.NOT.(+1.Ø).LE.DA(2)

as opposed to the following form which is incorrect:

.NOT.+1.Ø.LE.DA(2)

Logical and relational operators have the following priorities when precedence is not established by parentheses:

.GT. .GE. .LT. .LE. .EQ. .NE.
 .NOT.
 .AND.
 .OR.

Relational operators are evaluated in an expression from left to right subject to the priority rules above. In a logical expression, all arithmetic evaluation is performed prior to any logical evaluation. Thus, the logical expression:

.NOT.Z**2+Y*MA .GT. K-2 .OR. PA .AND. X .EQ. Y

is interpreted as:

(.NOT.(((Z**2) + (Y*MA)).GT.(K-2))).OR.(PA.AND.(X.EQ.Y))

Parentheses can be used within logical expressions to make an expression more readable or to indicate overriding of established priorities. For example, the following two expressions are evaluated differently:

A.AND.(B.OR.C)
 (A.AND.B).OR.C

CHAPTER 3

ASSIGNMENT STATEMENTS

There are three types of assignment statements:

1. Arithmetic assignment statements,
2. Logical assignment statements, and
3. ASSIGN statements.

A variable can be assigned a value at any point in a source program. During program execution, the most recent assignment determines the value of that variable in subsequent statements.

3.1 ARITHMETIC ASSIGNMENT STATEMENT

The arithmetic assignment statement provides the means by which the results of computations can be stored.

The format of the arithmetic assignment statement is:

variable = expression

where variable is any legal simple or subscripted variable name, expression is any legal arithmetic expression, and = is the replacement operator. The equal sign does not indicate equality as in an algebraic statement, but rather a replacement. The value of the expression is assigned to the variable name, regardless of any previous value of the variable. For example:

I = I+1

takes the current value of I, adds 1 to it, and uses the new value as the current value of I for any subsequent operations.

The expression to the right of the equal sign is evaluated and converted when necessary to conform to the type of the variable to the left of the equal sign. That is, if a real expression is assigned to an integer variable, the value of the expression is converted (truncated) to an integer before assignment. For example:

```

AND = Y*(X**2+Z)
I = I*N
X(J) = A(J)-B(J)
J = A**2+C

```

The expression to be assigned must be capable of yielding a value which conforms to the type attribute of the variable which is being assigned. Type conversions are performed in accordance with the rules stated for mixed mode expressions in Section 2.4.3 and Tables 2-1 and 3-1.

3.2 LOGICAL ASSIGNMENT STATEMENT

A logical assignment statement is of the form:

```
variable = expression
```

where variable is a predefined logical variable name or a logical array element name, expression is a logical expression, and = is the replacement operator. As in the arithmetic assignment statement, the equal sign indicates that the value of the expression is given to the variable specified. For example:

```

P = .TRUE.
VAL = X.GT.4.OR.B.EQ.X
A = R.LE.X**2

```

In the case where mixed mode arithmetic expressions are being logically compared, Table 3-1 shows the mode in which the comparison is effected. For example:

```
A = X**2 .GE. I
```

causes X**2 to be evaluated, the value of the integer variable I is converted to real format, and the comparison is made.

3.3 ASSIGN STATEMENT

The ASSIGN statement is of the form:

```
ASSIGN n TO ivar
```

Table 3-1

CONVERSION RULES FOR ASSIGNMENT STATEMENTS

Variable Type	E x p r e s s i o n T y p e				
	REAL	INTEGER, LOGICAL, Literal or OCTAL Constant	COMPLEX	DOUBLE PRECISION	LOGICAL*1
REAL	store X in V	Make X REAL and store in V	store REAL part of X in V, imaginary part lost	round to REAL and store in V, low-order portion lost	sign-extend byte to INTEGER, convert INTEGER to REAL and store in V.
INTEGER, LOGICAL, Literal or OCTAL Constant	truncate X to INTEGER and store in V	store X in V	truncate REAL portion of X to INTEGER and store in V, imaginary part lost.	truncate X to INTEGER and store in V	store X in low-order portion of V with sign bit extended through high-order portion of V.
COMPLEX	store X in REAL portion of V, imaginary portion = $\emptyset.\emptyset.$	make X REAL and store in real portion of V, imaginary portion = $\emptyset.\emptyset.$	store X in V	round X to REAL, store in real part of V, imaginary part = $\emptyset.\emptyset.$	sign-extend byte to INTEGER, convert integer to COMPLEX and store in V.
DOUBLE PRECISION	make X DOUBLE PRECISION with low-order portion = $\emptyset.$	make X REAL and store in high-order portion of V, low-order portion = $\emptyset.$	set V to real portion of X with low-order portion of V = $\emptyset.$, imaginary part lost.	store X in V	sign-extend byte to INTEGER, convert integer to Double Precision, store in V.
LOGICAL*1	convert real portion to INTEGER, store low-order 8 bits of INTEGER in byte.	store low-order 8 bits in V.	convert real portion to INTEGER, store low-order 8 bits of INTEGER in byte.	convert DOUBLE PRECISION portion to INTEGER, store low-order 8 bits of INTEGER in byte.	store X in V.

where: X indicates the expression as evaluated on the right of the =

V indicates the variable name to the left of the =
as in the statement: V=X

where n is a statement number and ivar is an integer variable.

Following an ASSIGN statement, a subsequently assigned GOTO statement (see Section 4.1.3) containing the integer variable ivar causes a transfer to the statement identified by the ASSIGN statement. The ASSIGN and associated assigned GOTO statements must appear in the same FORTRAN program unit. The statement number assigned must be that of an executable statement (assignment, I/O, or control statement).

An integer variable which is used in an ASSIGN statement must be redefined in an arithmetic assignment statement before it can be used in any context other than the GOTO statement (and after which it can no longer be used in the GOTO statement). For example:

```
ASSIGN 10 TO COUNT
```

associates the variable name COUNT with statement number 10, and after which the statement:

```
COUNT=COUNT+1
```

is considered invalid by the FORTRAN system. The previous statement can be performed, however, if preceded by the statement:

```
COUNT=20
```

which assigns the value of 20 to the variable COUNT. But the variable COUNT can no longer be used to refer to program line number 10. Incorrect usage of a variable defined in an ASSIGN statement causes a warning diagnostic to be printed by the Compiler.

CHAPTER 4

CONTROL STATEMENTS

4.1 GOTO STATEMENTS

GOTO statements allow program execution to jump from one section of code to another, either directly or based on the value of a variable.

There are three types of GOTO statements:

- a. Unconditional GOTO statements,
- b. Computed GOTO statements, and
- c. Assigned GOTO statements.

4.1.1 Unconditional GOTO Statement

The unconditional GOTO statement transfers control directly to the specified statement number. The form of this statement is:

GOTO n

where n is the statement number of an executable statement. For example:

GOTO 1000

When the above statement is encountered, program execution continues at the statement with the statement number 1000. An unconditional GOTO statement may appear anywhere in the executable part of a source program, except as the terminal statement of a DO loop.

4.1.2 Computed GOTO Statement

The computed GOTO statement allows the user to transfer control to one of several statement numbers, depending upon the value of a variable. The form of the statement is as follows:

GOTO($n_1, n_2, n_3, \dots, n_k$), ivar

where n_i are statement numbers and ivar is an integer variable or constant ($1 \leq \text{ivar} \leq k$). The comma between the statement number list and ivar is optional. Control is automatically transferred to the statement whose number is n_{ivar} in the list. For example:

GOTO (75,150,100,85,275)L

If L is equal to 4 when the statement is executed, control transfers to statement number 85 in the program. If L has a value less than 1 or greater than 5 at the time this statement is executed, an error message is printed. There is no restriction on other uses of the integer variable L within the program.

A computed GOTO statement can appear anywhere in the executable part of a source program except as the terminal statement of a DO loop.

4.1.3 Assigned GOTO Statements

The assigned GOTO statement allows the user to transfer control based on an integer variable name.

Assigned GOTO statements can take one of two forms:

GOTO K

or

GOTO K, ($n_1, n_2, n_3, \dots, n_k$)

where K is an integer variable name and n_i are statement numbers. Control is transferred to the statement whose number is currently associated with the variable K via an ASSIGN statement.

As described in Section 3.3, an ASSIGN statement defines an integer variable as a statement number. Thus, when the statement

ASSIGN 10 TO LOOP

is executed, the programmer can transfer control to statement

10 by inserting the statement:

```
GOTO LOOP
```

or, alternatively:

```
GOTO LOOP, (10, 20, 100)
```

which transfers control to whichever statement number is currently associated with LOOP. If the name LOOP is not defined as one of the listed statement numbers, an error diagnostic is printed.

4.2 IF STATEMENTS

IF statements allow program execution to jump from one section of code to another or conditionally execute a statement based on the value of an expression. There are two types of IF statements:

- a. Arithmetic IF statements, and
- b. Logical IF statements.

4.2.1 Arithmetic IF Statement

An arithmetic IF statement is of the form:

```
IF (expression)  $n_1, n_2, n_3$ 
```

where expression is an arithmetic expression in parentheses and n_1, n_2, n_3 represent the statement numbers to which control is transferred if the expression is less than, equal to, or greater than 0, respectively. All these statement numbers must be present. For example:

```
ALPHA = -7  
:  
:  
IF (ALPHA) 10, 20, 30
```

control transfers to statement number 10, since the expression

is less than zero. Complex valued expressions cannot be used in arithmetic IF statements.

4.2.2 Logical IF Statement

A logical IF statement is of the form:

IF (expression) statement

where expression is a logical expression in parentheses and statement is any executable FORTRAN statement other than another IF statement or a DO statement. If the expression has the value .TRUE. the statement is executed; otherwise, the next sequential statement is executed.

NOTE

Care should be taken in mixed mode comparisons or comparisons in which one or more values have undergone a type conversion. Mixed mode values to be compared are set to the more significant of the two types being compared. When this is done, repeating binary fractions do not repeat to the full precision of the new mode. For example, this may cause a real 0.1 not to equal a double precision 0.1.

Examples of logical IF statements are shown below:

```
IF (T.OR.S) X=Y+1
IF (Z.GT.X(K)) CALL SWITCH(S,Y)
IF (M.EQ.INDEX) GOTO 15
```

4.3 DO STATEMENT

The DO statement simplifies the coding of iterative procedures. The DO statement is of the form:

DO n i = m₁,m₂,m₃

where n is a statement number, i is a nonsubscripted integer

variable, and m_1, m_2 , and m_3 are positive integer constants or non-subscripted integer variables. If m_3 is not specified, it is understood to be 1.

Statements following the DO statement up to and including statement n are executed repeatedly for values of i starting with m_1 and incremented by m_3 until i is greater than m_2 .

The statements which are executed as a result of a DO statement are called the range of the loop. The variable i is called the index variable. The values m_1, m_2 , and m_3 are the initial, terminal, and increment values of the index, respectively.

When the DO statement is executed, its range is first executed with $i=m_1$. Subsequent iterations have $i=i+m_3$. A zero or negative m_3 value is not permitted.

The range of a DO loop is always executed at least once, regardless of the values of the limit and increment. After each execution of the range, the increment value is added to the value of the index and the result compared with the limit value. If the index value is not greater than the limit value, and integer overflow has not occurred, the range is executed again with the new index value.

For example:

```
DO 20 I = 5,100,2      (final iteration has I=99)
DO 100 M = 10,100,2    (final iteration has M = 100)
```

After the last execution of the range, control passes to the statement immediately following the loop. This is the normal exit from a DO loop. Exit can also be accomplished by the execution of a control statement within the range. (Control cannot be transferred into the range of a DO loop from outside that loop except in the case of an extended range as described below.)

The values of the limit and increment variables of the DO loop cannot be altered within the range of the DO statement. Within the range of a DO loop the index variable is available for use as an ordinary variable. When a statement transfers control outside the range of a DO loop, with a GOTO or IF, the index variable retains its current value and is available for use as a variable. The value of an index variable is not defined on normal exit from the DO loop.

The terminal statement of a DO loop cannot be a GOTO, DO, RETURN or arithmetic IF statement. A logical IF statement is allowed as the last statement of the range, provided it does not contain any of the statements already mentioned. For example:

```

      DO 5 K = 1,4
5      IF (X(K).GT.Y(L)) Y(K)=X(K)
6      ...

```

In this case, the loop ends when control passes to statement number 6. Statement 5 is executed four times whether the Y(K)=X(K) statement is executed or not. Note that if statement 5 were:

```

5      IF (X(K).GT.Y(L)) GOTO 10

```

improper execution may result.

The range of a DO statement may include other DO statements, provided the range of each DO statement is contained (nested) entirely within the range of the next outermost DO statement. That is, the range of any two DO statements must intersect completely or not at all. More than one DO loop within a nest of DO loops can end on the same terminal statement.

The range of a DO loop need not be merely a section of straight line code following the DO statement. A control statement which causes instructions elsewhere in the program to be executed is permissible as long as control returns to the range of the loop which it earlier left. The statements executed between the pair of control statements (leaving and returning to the DO loop) are called the extended range of the loop.

DO loops can be nested up to 10 levels deep. In calculating this depth, an implied DO in an I/O statement counts as one level, whose range is a single statement; and n implied DO loops within one I/O statement count as n levels whose ranges are all within the single statement. For example:

```
50 WRITE (5,250) (BUF,F(I),I=1,5), (DIFF(I),I=1,3)
```

is a statement containing two complete implied loops resulting in one level of nesting at a time. If the above statement were contained in an unnested DO loop, the maximum nesting level generated by line 50 is two.

NOTE

A common typing mistake is to replace a comma in a DO statement with a period. For example:

```
DO 5 I = 1.5
```

This statement appears to the FORTRAN Compiler as an arithmetic assignment statement:

```
DO5I = 1.5
```

4.4 CONTINUE STATEMENT

The CONTINUE statement is of the form:

```
CONTINUE
```

No processing is done when this statement is executed. It is used primarily as a target line number for control transfers, particularly as the last statement in the range of a DO loop. For example:

```
DO 7 K = START,END
  :
  :
  IF (X(K)) 22,13,7
  :
7  CONTINUE
```

a positive value of X(K) begins another execution of the range.

In this example, the CONTINUE statement provides a target address for the IF statement and ends the range of the DO loop.

The CONTINUE statement can be inserted anywhere in a program with no effect.

4.5 PAUSE STATEMENT

The PAUSE statement takes one of the following forms:

PAUSE
or
PAUSE n

where n is an octal constant from one to six digits long (up to 177777 and optionally preceded by the letter O). The PAUSE statement temporarily interrupts program execution. If present, the octal constant is printed on the error logging device following an A005 Monitor message when execution stops. For example:

A005 043521

where the octal constant was 043521 and

A005 000000

where no octal constant was specified.

Program execution can be resumed at the next executable FORTRAN statement by typing the CONTINUE command to the Monitor, followed by the RETURN key. Program execution can be terminated by typing the KILL command to the Monitor, followed by the RETURN key.

4.6 STOP STATEMENT

The STOP statement is of the form:

STOP
or
STOP n

where n is an octal constant from one to six digits long (optionally preceded by the letter O). The STOP statement causes all execution

of the current program to terminate and control returns to the DOS Monitor. The information message I35Ø is printed followed by the octal constant, if present. For example:

```
I35Ø 12415
```

where the octal constant was 12415 and

```
I35Ø ØØØØØØ
```

where no octal constant was specified. Use of STOP implies a call to the EXIT subroutine (See Section 6.5.5).

4.7 END STATEMENT

The END statement is of the form:

```
END
```

This statement is the necessary last statement in all main and subprograms. Program compilation is terminated when the END statement is encountered.

An END statement forces a call to the EXIT subroutine in a main program. In a function or subroutine subprogram it forces a return to the calling program if a RETURN statement is not present.

If a Compiler input file is exhausted before an END statement is encountered, a diagnostic is printed and an END statement forced.



1

1



1

1



CHAPTER 5

SPECIFICATION STATEMENTS

Specification statements are nonexecutable statements generally placed at the beginning of a program to inform the FORTRAN Compiler about storage allocation, variable types, subprograms, or data.

Specification statements must be placed in the program prior to the first usage of any variable they describe. Some specification statements determine storage allocation and must appear prior to any executable statements. Those statements which must precede all executable statements are called declaratives and include:

IMPLICIT
EXTERNAL
type declaration
COMMON
EQUIVALENCE
DIMENSION
DATA

The IMPLICIT declaration, if present, should occur first and the DATA declaration last, as these two statements may affect or be affected by, respectively, any other declaratives.

5.1 IMPLICIT STATEMENT

The IMPLICIT statement is of the form:

IMPLICIT type $a_1, a_2, (b_1-b_2), a_3, (b_3-b_4), \dots$

where type indicates the variable type being defined (see Section 5.2 for a complete list). The elements a_i represent single alphabetic characters separated by commas. The elements (b_n-b_m) represent a range of characters denoted by the first and last characters of the range and separated by a minus sign. A range of characters is always enclosed in parentheses. For example:

```
IMPLICIT INTEGER (A-D), (X-Z)
```

This statement instructs the FORTRAN Compiler that any variable name in the program beginning with the letters A,B,C,D,X,Y, or Z is an integer variable.

The IMPLICIT statement, if present in a routine, must precede all executable and specification statements (other than a SUBROUTINE or FUNCTION statement). The IMPLICIT statement becomes effective when it is encountered in the program. Previous specifications of variables having implicitly defined values will cause a diagnostic. For example:

```
COMMON Y1
IMPLICIT INTEGER (X-Z)
:
```

results in a W (warning) diagnostic.

The effect of this statement is to extend the basic rule that all names beginning with I,J,K,L,M or N indicate integer variables, all others being real variables. Any variable name which does not appear explicitly in a type declaration statement and whose first character is one of those listed in an IMPLICIT statement is classified according to the type appearing in that IMPLICIT statement.

The initial state of the FORTRAN Compiler is as though the statements:

```
IMPLICIT REAL (A-H), (O-Z)
IMPLICIT INTEGER (I-N)
```

appeared at the beginning of each program. This state is in effect unless a type declaration or IMPLICIT statement changes the interpretation of any variable name. Definition of a variable name in a type declaration overrides any type specification determined by an IMPLICIT statement.

Most of the FORTRAN Library function (such as FLOAT, IFIX, etc.) names follow the default type rules. That is, FLOAT produces a real result, IFIX produces an integer result, etc. Use of an IMPLICIT statement may change the default type, causing undesirable results. For example, in the program:

```
IMPLICIT INTEGER (A-Z)
REAL X
I=1
X=FLOAT(I)
END
```

the function FLOAT is treated as though it produced an integer result leading to incorrect values being generated. To have the FLOAT function return a real value in this context, an explicit type declaration should be included in the program before or after the IMPLICIT statement, as follows:

```
REAL FLOAT
```

Those library functions whose names do not follow the default type rules (those that return complex or double precision values) are recognized by the Compiler under any conditions. The data type of these names is not affected by an IMPLICIT statement but may be changed by an explicit type declaration. For example:

```
INTEGER CABS
```

This statement is likely to lead to incorrect operations at execution time if the intrinsic function CABS is called.

5.2 TYPE DECLARATION STATEMENT

A type declaration statement provides the Compiler with a description of the storage format of the variables defined, and is of the form:

```
type var1, var2, ...
```

where type indicated the name of the variable type being defined. Acceptable types include:

INTEGER	(1- or 2-word integer numbers)
INTEGER*2	(same as INTEGER)

Allocation of two bytes for one-word integers is effected only by the /ON switch. INTEGER*2 causes 4 bytes per element when the /ON switch is not set (see Section 12.2).

REAL	(2-word real numbers)
REAL*4	(same as REAL)
DOUBLE PRECISION	(4-word real numbers)
REAL*8	(same as DOUBLE PRECISION)
COMPLEX	(4-word values, having 2-word real part and 2-word imaginary part)
LOGICAL	(true = -1, false = 0)
BYTE	(8-bit format)
LOGICAL*1	(same as BYTE)

The terms $var_1, var_2, \dots$ indicate the variable name(s) which are to be allocated storage space as the data type specified.

Type declarations must precede all executable statements.
For example:

```
COMPLEX RET,X,TST
BYTE IAMB,LEN,R(20),S(20)
```

In the first statement three variable names are specified as complex variables. In the second statement, two variable names are specified as 8-bit byte variables, and the arrays R and S are dimensioned as having 20 byte-sized elements each.

Type declarations can be used to define dimension specifications for arrays. However, an array named in a type declaration can be dimensioned only in one specification statement. Adjustable arrays in subprograms can also be defined in type declarations.

If a variable name is defined in a type declaration, the type so specified overrides any type specification determined by the initial letter of the variable. The priority for defining a variable type is, therefore, as follows:

- a. type declarations
- b. IMPLICIT variable type
- c. real or integer depending on initial letter
(where I-N indicates an integer).

5.3 DIMENSION STATEMENT

The DIMENSION statement is of the form:

DIMENSION $S_1, S_2, S_3, \dots$

where S is an array specification of the form

var(n_1)
var(n_1, n_2)
or var(n_1, n_2, n_3)

where var is the array variable name and n_1 , n_2 , and n_3 are unsigned integer constants (or, optionally, variables in the case of adjustable array dimensions in subprograms) indicating the maximum size of the array in each of the three possible dimensions.

An array can have one, two or three dimensions. For example:

DIMENSION A(20), B(10,10), C(5,5,5)

Where more than one array is defined in a single DIMENSION statement, the array specifications are separated by commas.

DIMENSION statements precede all executable statements.

The appearance of the DIMENSION statement causes the FORTRAN Compiler to allocate space for the appropriate number of variables. Storage allocation can also be performed by a COMMON statement or any type declaration statement. For example:

```
COMMON X(10,4),Y,Z
INTEGER A(7,32)
```

Unless an array is specified in a type declaration statement, the type of the array is determined by the first letter of the array variable name.

An error message is always generated at run-time if an array subscript is less than one. Optionally, using the /CK switch option (see Section 12.2), the system will check to see that array subscripts do not exceed the maximum size specified.

A subprogram can use adjustable arrays, the size of which may depend upon the parameters of the subprogram call (see Sections 6.4.1 and 6.4.2).

5.3.1 Array Storage

Arrays of one, two or three dimensions are each stored as a linear sequence of values in memory. Arrays are always stored in memory column-by-column. Therefore, the n^{th} element of a one dimensional array is $A(n)$.

In order to find the linear element of a two-dimensional array $A(I,J)$ where a given element is $A(i,j)$, the formula:

$$n = i + I * (j - 1)$$

is evaluated. $A(i,j)$ is stored at the n th element.

1-Dimensional Array A(10)

A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	A(7)	A(8)	A(9)	A(10)
------	------	------	------	------	------	------	------	------	-------

2-Dimensional Array B(5,5)

1	B(1,1)	6	B(1,2)	11	B(1,3)	16	B(1,4)	21	B(1,5)
2	B(2,1)	7	B(2,2)	12	B(2,3)	17	B(2,4)	22	B(2,5)
3	B(3,1)	8	B(3,2)	13	B(3,3)	18	B(3,4)	23	B(3,5)
4	B(4,1)	9	B(4,2)	14	B(4,3)	19	B(4,4)	24	B(4,5)
5	B(5,1)	10	B(5,2)	15	B(5,3)	20	B(5,4)	25	B(5,5)

B(3,1) is the third storage element in sequence
B(3,4) is the eighteenth storage element in sequence

3-Dimensional Array C(5,5,5)

101	C(1,1,5)	106	C(1,2,5)	111	C(1,3,5)	116	C(1,4,5)	121	C(1,5,5)
102	C(2,1,5)	107	C(2,2,5)	112	C(2,3,5)	117	C(2,4,5)	122	C(2,5,5)
76	C(1,1,4)	81	C(1,2,4)	86	C(1,3,4)	91	C(1,4,4)	96	C(1,5,4)
77	C(2,1,4)	82	C(2,2,4)	87	C(2,3,4)	92	C(2,4,4)	97	C(2,5,4)
51	C(1,1,3)	56	C(1,2,3)	61	C(1,3,3)	66	C(1,4,3)	71	C(1,5,3)
52	C(2,1,3)	57	C(2,2,3)	62	C(2,3,3)	67	C(2,4,3)	72	C(2,5,3)
26	C(1,1,2)	31	C(1,2,2)	36	C(1,3,2)	41	C(1,4,2)	46	C(1,5,2)
27	C(2,1,2)	32	C(2,2,2)	37	C(2,3,2)	42	C(2,4,2)	47	C(2,5,2)
1	C(1,1,1)	6	C(1,2,1)	11	C(1,3,1)	16	C(1,4,1)	21	C(1,5,1)
2	C(2,1,1)	7	C(2,2,1)	12	C(2,3,1)	17	C(2,4,1)	22	C(2,5,1)
3	C(3,1,1)	8	C(3,2,1)	13	C(3,3,1)	18	C(3,4,1)	23	C(3,5,1)
4	C(4,1,1)	9	C(4,2,1)	14	C(4,3,1)	19	C(4,4,1)	24	C(4,5,1)
5	C(5,1,1)	10	C(5,2,1)	15	C(5,3,1)	20	C(5,4,1)	25	C(5,5,1)
78	C(1,1,3)	83	C(1,2,3)	88	C(1,3,3)	93	C(1,4,3)	98	C(1,5,3)
79	C(2,1,3)	84	C(2,2,3)	89	C(2,3,3)	94	C(2,4,3)	99	C(2,5,3)
53	C(1,1,2)	58	C(1,2,2)	63	C(1,3,2)	68	C(1,4,2)	73	C(1,5,2)
54	C(2,1,2)	59	C(2,2,2)	64	C(2,3,2)	69	C(2,4,2)	74	C(2,5,2)
28	C(1,1,1)	33	C(1,2,1)	38	C(1,3,1)	43	C(1,4,1)	48	C(1,5,1)
29	C(2,1,1)	34	C(2,2,1)	39	C(2,3,1)	44	C(2,4,1)	49	C(2,5,1)
3	C(3,1,1)	8	C(3,2,1)	13	C(3,3,1)	18	C(3,4,1)	23	C(3,5,1)
4	C(4,1,1)	9	C(4,2,1)	14	C(4,3,1)	19	C(4,4,1)	24	C(4,5,1)
5	C(5,1,1)	10	C(5,2,1)	15	C(5,3,1)	20	C(5,4,1)	25	C(5,5,1)

C(1,3,2) is the 36th storage element in sequence.
C(1,1,5) is the 101st storage element in sequence.

Figure 5-1

Array Storage

In the three-dimensional array $A(I,J,K)$ where a given element is $A(i,j,k)$, an individual element is stored at the n th element where n is:

$$n = i + I*(j-1) + I*J*(k-1)$$

It is sometimes desirable to equivalence a two- or three-dimensional array to a one-dimensional array. The above formulas are useful for determining the algorithms for manipulating the arrays after such a change. However, for any given array, elements of that array must be referenced using the number of subscripts specified in the defining statement (in some contexts an entire array can be referenced by indicating the array name only).

5.4 COMMON STATEMENT

The COMMON statement causes the specified variables to be stored in an area of memory available to other programs and subprograms. Common block areas can be of any size and any number of them can exist within the space available on the system. One unnamed common block is allowed, and is known as blank common. The variables are placed in the individual common blocks in the order in which they appear in the block description.

The COMMON statement takes the form:

```
COMMON /block1/a,b,c/block2/d,e,f
```

where blockn is a one-to six-character alphanumeric name (the first character of which must be alphabetic), delimited by slashes, and considered to be the block name. The sequences a,b,c and d,e,f represent the names of the variables assigned to the common block(s).

The form:

```
COMMON a,b,c,...,n
```

is used to indicate the blank common block.

One or more common blocks can be defined in the same COMMON statement.

For example:

```
COMMON /A/X,Y,Z/B/L,M,N
```

This statement defines two common blocks, each containing three variables.

Where blank common is the first common block being described in the COMMON statement, no block name or specification is necessary. Subsequent block names in the same COMMON statement are delimited by slashes. However, when elements of blank common are indicated other than in the first position in a COMMON statement, two consecutive slashes (any separating spaces are ignored) are used. For example:

```
COMMON /R/X,Y//B,C,D
COMMON B,C,D/R/X,Y
```

The two COMMON statements above are identical in effect.

All COMMON statements within a program unit must precede all executable statements.

A common variable cannot appear in more than one common block in a single program unit. However, since labeled common block names are used only by the Compiler, and are not present in the functional program at run time, such block names can be used as variable or subroutine names within the program.

Block entries are linked sequentially throughout the program, beginning with the first COMMON statement. For example, the statements:

```
COMMON/A/ALPHA/B/BET,GAM/C/ET
COMMON/B/DEL,EP,ZE/C/THE,IO/D/KA,LA
```

have the same effect as the statement:

```
COMMON/A/ALPHA/B/BET,GAM,DEL,EP,ZE/C/ET,THE,IO/D/KA,LA
```

Storage allocation for blocks of the same name begins at the same location for all program units linked together in a single load module by Link-11. For example, if a program contains the statement:

```
COMMON A,B/R/X,Y,Z
```

as its first COMMON statement, and a subprogram contains the statement:

```
COMMON /R/U,V,W//D,E
```

as its first COMMON statement, the quantities represented by X and U, by Y and V, and by Z and W (provided they are of corresponding word lengths) are stored at the same locations in common block R; and a similar correspondence holds for the quantities A and D, and B and E in the blank common block. Responsibility for the values of the variables used and the correspondence of word lengths of the variables rests with the programmer.

The size of a common block is the sum of the storage required for all elements introduced through COMMON and EQUIVALENCE statements. Where the size of the various common blocks referenced by several program units in a single load module differs, the length of each common block is resolved by Link-11 to the length of the longest block of the given name declared by any program unit.

Care is required when creating an overlay system to insure that the longest block of a given name is declared within the resident section rather than in any of the overlay files (see Section 7.2).

The variable names which follow the block name indicate scalar or array variables assigned to that block. Where an unsubscripted array name appears in a COMMON statement, the array must have been previously defined in a DIMENSION or type declaration statement. Where a subscripted array name appears in a COMMON statement and the array has not been defined, the array is defined to the indicated size. Where the array has been dimensioned previously, an error diagnostic is printed. For example:

```
DIMENSION T(10)
COMMON ABS(5,10,5),T
```

The COMMON statement defines the array ABS and allocates storage space for that array in blank common. All elements of the array T are also stored in blank common. (Arrays in common storage are stored according to the rules specified in Section 5.3.1.)

5.5 EQUIVALENCE STATEMENT

The EQUIVALENCE statement is of the form:

EQUIVALENCE (var₁,var₂,...), (var_i,var_j,...),...

where var indicates a variable name used in the current program. Each of the variables within a set of parentheses is assigned the same storage location. For example:

EQUIVALENCE (LAN,MAR)

specifies that the variables LAN and MAR are stored in the same location, and have the same value. Where variables being equivalenced are not of the same type, the equivalencing operation assumes that each variable begins in the same word (see the description of storage formats in Appendix B).

EQUIVALENCE statements must precede all executable statements.

The equivalence relationship is transitive; that is, the two statements below have the same effect:

EQUIVALENCE (A,B), (B,C)
EQUIVALENCE (A,B,C)

The EQUIVALENCE statement is used to conserve core storage during execution or to obtain a portion of a numeric value in a different format. Rather than reserving space for a variable which is only used in a portion of the program, a later variable can be equivalenced to use this space when the earlier value is no longer of interest. A statement such as the following:

EQUIVALENCE (TRI,ISO)

allows ISO to reference the first storage location of the real variable TRI.

To make full use of the space required by a real variable, that real variable can be equivalenced to two integer variables (assuming use of the /ON switch which causes one-word integers to be used throughout the program). Similar techniques can be used to equivalence two real, or two to four integer, variables to complex or double-precision variables. For example:

```
DIMENSION N(2)
EQUIVALENCE (TRI,N)
```

These two statements equivalence the first word of TRI to N(1) and the second word of TRI to N(2), assuming use of the /ON switch.

5.5.1 Equivalencing Array Variables

The subscripts of any array variables appearing in an EQUIVALENCE statement must be integer constants. For example:

```
EQUIVALENCE (X,A(5)),(BET(2,2),Y(2,4,1),NET)
```

A previously dimensioned array name used in an EQUIVALENCE statement without subscripts implies reference to the first element of the array. For example:

```
DIMENSION Y(20)
EQUIVALENCE (Y,X)
```

has the same effect as:

```
DIMENSION Y(20)
EQUIVALENCE (Y(1),X)
```

Where a subscript is present in an EQUIVALENCE statement, the array element specification must contain an integer constant for each of the array dimensions as originally defined. No default value is supplied for missing dimension elements, and a diagnostic is generated. For example:

```
DIMENSION REAL(2,10,10)
EQUIVALENCE (REAL(1,1),TRAN)
```

is illegal and would result in an error message. (See also the description of array storage in Section 5.3.1.)

5.5.2 EQUIVALENCE and COMMON Interaction

Variables can appear in both COMMON and EQUIVALENCE statements. However, two quantities in common cannot be made equivalent to each other. Quantities placed in a common block by an EQUIVALENCE statement may cause the end of the common block to be extended. For example:

```
COMMON/R/X,Y,Z
DIMENSION A(4)
EQUIVALENCE (A,Y)
```

causes the common block R to extend from X to A(4) as follows:

```
X
Y = A(1)
Z = A(2)
A(3)
A(4)
```

The common block has been extended by two elements. No EQUIVALENCE statements are allowed to extend the beginning of a common block. For example, the sequence:

```
COMMON/R/X,Y,Z
DIMENSION A(4)
EQUIVALENCE (X,A(4))
```

is not permitted since it would require A(1), A(2) and A(3) to extend beyond the starting location of common block R.

An EQUIVALENCE statement is not permitted to violate any previous EQUIVALENCE storage assignments or extend an array beyond its maximum predefined dimensions. For example:

```
DIMENSION A(10),B(10)
EQUIVALENCE (A(10),B(1)), (A(5),B(3))
EQUIVALENCE (A(5,10),B(5))
```

Both of the preceding EQUIVALENCE statements are unacceptable.

5.5.3 EQUIVALENCE and BYTE Arrays

In using EQUIVALENCE with items in a BYTE (LOGICAL*1) array, only those items aligned on a word boundary can be specified in an EQUIVALENCE statement. Individual BYTE variables are always aligned on word boundaries. In a BYTE array, the elements having odd subscripts are aligned on word boundaries. For example:

```
BYTE  A,B(9)
REAL  X
```

correct: EQUIVALENCE(A,X,B(3))

incorrect: EQUIVALENCE(A,X,B(4))

5.6 EXTERNAL STATEMENT

The EXTERNAL statement is of the form:

```
EXTERNAL a,b,c,...
```

where a,b,c,... represent the names of FUNCTION or SUBROUTINE subprograms to be used as parameters to a subprogram call.

FUNCTION and SUBROUTINE subprogram names can be used as arguments of subprograms. When used in this manner, subprogram names are distinguished from ordinary variables by their appearance in an EXTERNAL statement. For example:

```
EXTERNAL SIN,COS
CALL TRIGF(SIN,1.5,ANSWER)
CALL TRIGF(COS,18.7,ANSWER)
:
END
```

```
SUBROUTINE TRIGF(FUNC,ARG,ANSWER)
:
ANSWER = FUNC(ARG)
RETURN
END
```

The EXTERNAL statement must appear before any executable statements in any program or subprogram. This statement is used in programs or subprograms using the name of a function or subroutine as a parameter in a subprogram call. This statement does not take a line number and should not contain any name previously defined.

5.7 DATA STATEMENT

The DATA statement is of the form:

DATA var/value/

where var is a list of variable names, and value is an ordered list of numeric or string values which are assigned to the respective elements of the variable list.

The DATA statement is used to supply initial or constant values for variables. DATA statements must precede all executable statements and must follow all other declaratives. Elements of the variable and value lists are separated by commas. More than one set of variable and value lists can be contained in a single DATA statement. For example:

```
DATA A,B,C/3.4,5.9,7./  
DATA I,J/7,3/X,Y,Z/43.5,2.01,8.4/
```

Dummy arguments may not appear in the variable list. Whenever several variables are to be assigned the same value, the item in the value list can be preceded by an integer constant indicating the number of repetitions of that variable. The integer and data values are separated by an asterisk. For example:

```
DATA A,B,C/3*3.17/
```

The three real variables are each assigned the value 3.17.

The specified values are compiled into the object program and are assumed by the variables when program execution begins. Such values can also be provided for labelled common block

variables by means of a BLOCK DATA subprogram (see Section 6.4.8). Variables in a labeled common block can only be defined in a BLOCK DATA subprogram. Variables in the blank common block cannot be initialized.

Variables can also be subscripted array elements or unsubscripted array names. When the unsubscripted name of a predefined array is given in the list, data values must be specified for every element of the array. Data elements are stored in the array in the order used for data transmission and storage of arrays; that is, in order of the increasing subscripts with the first subscript varying most rapidly (column-by-column storage, see Section 5.3.1). Allocation of data to an array stops when:

- a. The data (value) list is exhausted; or
- b. Data items have been allocated to the entire array. In which case additional data items are allocated to any additional items in the variable list.

For example:

```
DATA X,Y,Z/'A','BCDE','FGHIJKL'//
```

produces in memory:

X+1 Blank	X A
X+3 undefined	X+2 undefined
Y+1 C	Y B
Y+3 E	Y+2 D
Z+1 G	Z F
Z+3 I	Z+2 H

The data items following each list of variables must have a one-to-one correspondence with the variables of the list, and must agree in type, since each item of the data specifies the value given to the corresponding variable.

Data items assigned may be numeric, Hollerith, octal, hexadecimal, or logical constants. For example:

```
DATA ALPHA,BETA,IVAL/5.,16.E-2,.TRUE./
```

specifies the value 5.0 for ALPHA, the value 0.16 for BETA, and the value -1 for IVAL. Any item of data may be preceded by an integer constant followed by an asterisk. This notation indicates that the item is to be repeated. Example:

```
DATA A(1),A(2),A(3)/3*0./
```

specifies the value 0.0 for array elements A(1) to A(3). For example:

```
DIMENSION A(2,2),B(3)
DATA A,B/2*1.0,3*2.0,3.0,4.0/
```

will initialize:

```
A(1,1), and A(2,1) to 1.0
A(1,2), A(2,2) and B(1) to 2.0
B(2) to 3.0, and B(3) to 4.0
```

As another example:

```
DIMENSION R(4),I(3),L(5)
DATA L,I,R/4*0177777,2*'AB', 2*4,4*1.0/
```

results in:

```
L(1) to L(4) = 177777 (octal)
L(5),I(1) = 'AB' (Hollerith)
I(2),I(3) = 4
R(1) to R(4) = 1.0
```



CHAPTER 6

FORTRAN FUNCTIONS AND SUBROUTINES

6.1 PROCEDURES

FORTRAN contains provisions for the creation and use of coded procedures. Procedures consist of FORTRAN (or assembly language) routines which perform operations for, and independently of, the main FORTRAN program. Procedures perform operations which may be required more than once in the course of a program and which might otherwise require repetition of code within a program. Their use, then, generally saves programming effort and reduces storage used.

There are two main types of procedures: functions and subroutines which include the following:

- 1) library functions (provided by the FORTRAN OTS)
- 2) statement functions (one-line, user defined, function definitions)
- 3) external functions (user-defined function subprograms)
- 4) system subroutines (provided by the FORTRAN OTS)
- 5) subroutines (user-defined subroutine subprograms)

Library functions and system subroutines are provided by the FORTRAN Object Time System and need only be referenced in or called by the user program. (See Sections 6.2 and 6.5 for a description of the available routines.) External functions and subroutines can be compiled independently of the main program and are linked to the main program with the PDP-11 Linker.

The term subprogram refers to both external functions and subroutines. Subprograms can communicate with the main program and among themselves by means of parameters specified in the subprogram call or by means of variables stored in common blocks. Calls to subprograms may not be recursive; that is, a call to any subprogram cannot result in subsequent calling of that subprogram, directly or indirectly.

6.2 FORTRAN LIBRARY FUNCTIONS

The FORTRAN library functions are described in Table 6-1. In order to use a library function in any routine, it is only necessary to use the function in a FORTRAN statement. For example:

$$R = \text{ABS}(X-1)$$

causes the absolute value of $X-1$ to be calculated and assigned to the variable R .

The type of each library function is predefined as described in Table 6-1. Arguments passed to these functions may consist of subscripted or simple variable names, constants, arithmetic expressions or other intrinsic functions. Arguments to these functions must correspond to the type indicated in Table 6-1. If the argument types do not match, no conversion is performed, and no diagnostic is given. See also Section 5.1 for the effect of the IMPLICIT statement on function values.

Details on the algorithms used in the individual library functions are contained in Appendix D.

6.3 ARITHMETIC STATEMENT FUNCTIONS

An arithmetic statement function is used to define one-line functions for the user. These functions can be referenced only by the program unit in which they are defined.

To define an arithmetic statement function, the following form is used:

$$\text{name}(\text{arg}_1, \dots) = \text{expression}$$

where name is the function name, which must be a legal FORTRAN variable name. Function type is determined either implicitly by the type of the function name or explicitly through the indication of a type specification. The term $(\text{arg}_1, \dots)$ represents the list of dummy variables used in the function definition; at least one argument must be present; expression is the function definition.

The definition of an arithmetic statement function must occur after all declaratives (specification statements) and before any executable statements.

Table 6-1

FORTRAN Library Functions

Form	Definition	Argument Type	Result Type
ABS(X)	Real absolute value	Real	Real
IABS(I)	Integer absolute value	Integer	Integer
DABS(X)	Double precision absolute value	Double	Double
CABS(Z)	Complex to Real, absolute value where $Z = (x, y)$ $CABS(Z) = (x^2 + y^2)^{1/2}$	Complex	Real
FLOAT(I)	Integer to Real conversion	Integer	Real
IFIX(X)	Real to Integer conversion IFIX(X) is equivalent to INT(X)	Real	Integer
SNGL(X)	Double to Real conversion	Double	Real
DBLE(X)	Real to Double conversion	Real	Double
REAL(Z)	Complex to Real conversion obtain real part	Complex	Real
AIMAG(Z)	Complex to Real conversion, obtain imaginary part	Complex	Real
CMPLX(X, Y)	Real to Complex conversion $CMPLX(X, Y) = X + iY$	Real	Complex
Truncation functions return the sign of the argument * largest integer $\leq arg $			
RAINT(X)	Real to Real truncation	Real	Real
RINT(X)	Real to Integer truncation	Real	Integer
RDINT(X)	Double to Integer truncation	Double	Integer
Remainder functions return the remainder when the first argument is divided by the second.			
RAMOD(X, Y)	Real remainder	Real	Real
MOD(I, J)	Integer remainder	Integer	Integer
DMOD(X, Y)	Double precision remainder	Double	Double
Maximum value functions return the largest value from among the argument list; ≥ 2 arguments.			
AMAX0(I, J, ...)	Real maximum from Integer list	Integer	Real
AMAX1(X, Y, ...)	Real maximum from Real list	Real	Real
MAX0(I, J, ...)	Integer maximum from Integer list	Integer	Integer
MAX1(X, Y, ...)	Integer maximum from Real list	Real	Integer
DMAX1(X, Y, ...)	Double maximum from Double list	Double	Double
Minimum value functions return the smallest value from among the argument list; ≥ 2 arguments.			
AMIN0(I, J, ...)	Real minimum of Integer list	Integer	Real
AMIN1(X, Y, ...)	Real minimum of Real list	Real	Real
MIN0(I, J, ...)	Integer minimum of Integer list	Integer	Integer
MIN1(X, Y, ...)	Integer minimum of Real list	Real	Integer
DMIN1(X, Y, ...)	Double minimum of Double list	Double	Double
The transfer of sign functions return (sign of the first argument) * (absolute value of second argument).			
SIGN(X, Y)	Real transfer of sign	Real	Real
ISIGN(I, J)	Integer transfer of sign	Integer	Integer
DSIGN(X, Y)	Double precision transfer of sign	Double	Double

Table 6-1 (Continued)

Form	Definition	Argument Type	Result Type
Positive difference functions return the first argument minus the minimum of the two arguments.			
DIM(X,Y)	Real positive difference	Real	Real
IDIM(I,J)	Integer positive difference	Integer	Integer
Exponential functions return the value of e raised to the argument power.			
EXP(X)	e^X	Real	Real
DEXP(X)	e^X	Double	Double
CEXP(Z)	e^Z	Complex	Complex
ALOG(X)	Returns $\log_e(X)$	Real	Real
ALOG10(X)	Returns $\log_{10}(X)$	Real	Real
DLOG(X)	Returns $\log_e(X)$	Double	Double
DLOG10(X)	Returns $\log_{10}(X)$	Double	Double
CLOG(Z)	Returns $\log_{10}$ of complex argument	Complex	Complex
SQRT(X)	Square root of Real argument	Real	Real
DSQRT(X)	Square root of Double precision argument	Double	Double
CSQRT(Z)	Square root of Complex argument	Complex	Complex
SIN(X)	Real sine	Real	Real
DSIN(X)	Double precision sine	Double	Double
CSIN(Z)	Complex sine	Complex	Complex
COS(X)	Real cosine	Real	Real
DCOS(X)	Double precision cosine	Double	Double
CCOS(Z)	Complex cosine	Complex	Complex
TANH(X)	Hyperbolic tangent	Real	Real
ATAN(X)	Real arc tangent	Real	Real
DATAN(X)	Double precision arc tangent	Double	Double
ATAN2(X,Y)	Real arc tangent of (X/Y)	Real	Real
DATAN2(X,Y)	Double precision arc tangent of (X/Y)	Double	Double
CONJG(Z)	Complex conjugate, if $Z=X+i*Y$ $CONJG(Z)=X-i*Y$	Complex	Complex
RAN(I,J)	Returns a random number of uniform distribution over the range 0 to 1. I and J must be integer variables and should be set initially to 0. Resetting I and J to 0 regenerates the random number sequence. Alternate starting values for I and J will generate different random number sequences. See also Section 6.5.4.	Integer	Real

Sine and cosine functions take arguments expressed in radians.

An arithmetic statement function definition must be contained in a single statement. The expression which defines the function can include dummy arguments, ordinary variables, intrinsic functions, external functions, and previously defined internal functions. For example:

$$\text{ACOSH}(X,A) = (\text{EXP}(X/A) + \text{EXP}(-X/A))/2.$$

is an acceptable arithmetic statement function. X and A are dummy arguments and may be used in the body of the user program. When the function is referenced, the real parameter of the function call, which replaces A and X in the function definition, is used to evaluate the function. All function definitions of this type must precede the first executable statement of the program in which they are referenced and follow the last specification statement appearing in the program.

An arithmetic statement function cannot include references to an array whose name is a dummy parameter. For example:

```
DIMENSION X(10)
F(X) = Y+X(1)+X(2)+X(3)
```

would generate a function reference to the array X and not an array reference. No diagnostic is issued.

However, in the following definition:

$$G(Y) = F(X) + X(Y+2)$$

X is legal as an array name reference since it is not a formal parameter.

6.4 EXTERNAL SUBPROGRAMS

6.4.1 Subprogram Parameters

Variable names and identifiers used within subprograms are independent of variable names in other program units. Those variable names and identifiers used in the definition of a subprogram and as parameters in the FUNCTION or SUBROUTINE statements are called dummy arguments. The rules concerning dummy arguments are identical for both function and subroutine subprograms.

Subprograms are written much like a FORTRAN main program. No conflicts arise where a single variable name is used in more than one subprogram or in both a subprogram and the main program. The first statement of a subprogram must be either a FUNCTION or a SUBROUTINE statement with a list of parameters. This list of parameters contains the dummy arguments of the subprogram. When the subprogram is called, dummy arguments are replaced with the actual arguments, as specified in the calling statement.

A function reference returns a single value. The parameters to a function reference are arguments needed by the function to obtain the returned value. The value of the function parameters can be altered within the function to return additional values. A subroutine call may return zero, one or more values to the calling program.

Actual arguments which are passed to a subprogram can be any of the following:

- arithmetic or logical expressions
- alphanumeric strings
- array names or elements
- subprogram names

(Expressions may contain numeric values, simple variables, array elements and library functions, alone or separated by operators.)

Actual and dummy arguments must agree in order, type and number. No type conversion is performed where a difference in data type occurs and no diagnostic is given.

Dummy arguments which represent array names must appear within the subprogram either in a DIMENSION statement or in one of the type statements that contain dimension information. Dimensions within a subprogram must not exceed the dimensions of the corresponding array(s) in the calling program. Dimensions of array arguments for a subprogram can be passed to the subprogram as arguments and optionally used to specify adjustable dimensions for array name arguments (see Section 6.4.2).

Dummy arguments must not appear in EQUIVALENCE, DATA or COMMON statements within a subprogram.

Function and subroutine subprogram names may be used as the actual arguments of subprograms. If a function or subroutine name is passed as an argument, the name must appear in an EXTERNAL statement in the calling program unit. The EXTERNAL statement identifies the argument as a subprogram name rather than a variable name.

For example, the subprograms TRAN1 and TRAN2 are:

```
REAL FUNCTION TRAN1(FUNC,ARG,N)
TRAN1= N*FUNC(ARG)
RETURN
END
```

```
REAL FUNCTION TRAN2(VAL,N)
TRAN2= N*VAL
RETURN
END
```

The two function subprograms each result in the same value being assigned to X, given that the call to the first is expressed as:

```
EXTERNAL SIN
.
.
.
X=TRAN1(SIN,25.0,N)
```

and the call to the second is expressed as:

```
X=TRAN2(SIN(25.0),N)
```

In the case of the first function, the capability of generating the function which replaces the dummy variable A is passed to the subprogram. In the case of the second function, a single value is passed to the subprogram as an argument which consists of the value of a library function expression.

6.4.2 Adjustable Dimensions

In order to obtain maximum flexibility from a subprogram, the user may wish to vary the dimensions of an array with each subprogram call. A subprogram may contain adjustable arrays providing it references an array which has been allocated storage space by the calling program and passed as a parameter.

In this case, both the array name and subscript values must be expressed as dummy arguments in the FUNCTION or SUBROUTINE statement beginning the subprogram. The array name (which may be any data type) and the subscript variables (which must be simple integer variables) are used to define the array within the subprogram as follows:

```
SUBROUTINE TRAN(A,I,J,RET,DET)
  DIMENSION A(I,J)
  .
  .
  .
  RETURN
```

The dummy array must not exceed the dimensions of the main program array, but may be smaller.

The arguments representing the array dimensions should be defined prior to the subprogram call and cannot be redefined or made undefined within the subprogram. The dimension specifications for an adjustable array must be supplied as a parameter in the subprogram call. (This implies that the dimension specifications may not reside in COMMON.)

6.4.3 FUNCTION Subprograms

A FUNCTION subprogram is a computational procedure which returns primarily a single value; functions can also return values through their arguments. A FUNCTION subprogram is invoked by using the function name with associated arguments in an arithmetic expression. A FUNCTION subprogram begins with a FUNCTION statement and ends with an END statement. Control is returned to the calling statement by means of one or more RETURN statements. (The END statement acts as an implied RETURN statement.) The mode of the function (the type of value returned) is determined by the name of the function or a type designation inserted into the initial FUNCTION statement.

The FUNCTION statement is of the form:

FUNCTION name(arg₁,...,arg_n)

or

type FUNCTION name(arg₁,...,arg_n)

This statement declares the program which follows to be a FUNCTION subprogram. The function name can be any legal FORTRAN variable name which is not used as a dummy argument. This name must not appear in any non-executable statement in the calling program or function subprogram, other than as a scalar variable in a type declaration. The function name must appear, at some point in the subprogram, as a scalar variable and be assigned a value. The value assigned to the function name is the value returned by the function reference in the calling program.

The argument list in the FUNCTION statement must consist of at least one dummy argument. Dummy arguments follow the rules established for subprogram parameters as described in Sections 6.4.1 and 6.4.2.

The type of the value returned by the function is normally determined to be either INTEGER or REAL according to the default type rule applied to the function name. However, a different type can be defined for the returned value by placing a type designation term before the FUNCTION statement or placing a type declaration statement within the subprogram.

A FUNCTION statement must not have a statement number. The only FORTRAN statements which may not appear in a FUNCTION subprogram are SUBROUTINE, BLOCK DATA and another FUNCTION statement.

An example of a FUNCTION subprogram is shown in Figure 6-1.

6.4.4 SUBROUTINE Subprograms

A SUBROUTINE subprogram is a computational procedure which returns zero, one or more values to the calling program. Values and data types are associated with the subroutine parameters, but

```

C      FRAGMENTARY FORTRAN PROGRAM
C      FOR ADJUSTABLE ARRAYS
C
0001    REAL X(10,10), Y(25,25)
C      .
C      .
C      .
0002    XT=TRACE(X,10)
0003    YT=TRACE(Y,25)
C      .
C      .
C      .
0004    END

```

ROUTINES CALLED:
TRACE

BLOCK	LENGTH
MAIN.	1500 (005670)*

```

**COMPILER ----- CORE**
      PHASE      USED  FREE
DECLARATIVES 00366 17629
EXECUTABLES  00446 17549
ASSEMBLY     00881 19977

```

```

0001    REAL FUNCTION TRACE(A,N)
C
C      COMPUTE THE TRACE OF N * N MATRIX A
C
0002    DIMENSION A(N,N)
C
0003    SUM=0.0
0004    DO 10 I=1,N
0005    10    SUM = SUM + A(I,I)
0006    TRACE=SUM
0007    RETURN
0008    END

```

BLOCK	LENGTH
TRACE	82 (000244)*

```

**COMPILER ----- CORE**
      PHASE      USED  FREE
DECLARATIVES 00366 17629
EXECUTABLES  00446 17549
ASSEMBLY     00929 19929

```

Figure 6-1

Example FUNCTION Subprogram

not with the subroutine name. A SUBROUTINE subprogram begins with a SUBROUTINE statement and ends with an END statement. Control is returned to the calling statement by means of one or more RETURN statements.

The SUBROUTINE statement is of the form:

SUBROUTINE name

or

SUBROUTINE name(arg₁,...,arg_n)

This statement declares the program which follows to be a SUBROUTINE subprogram. The subroutine name can be any legal FORTRAN variable name, but should not be used as a variable name within the calling program or subprogram itself, or as the name of a labelled COMMON block used in the same load module.

The argument list in the SUBROUTINE statement consists of any number of dummy arguments (or none). Dummy arguments follow the rules established for subprogram parameters as described in Sections 6.4.1 and 6.4.2. Dummy arguments to a SUBROUTINE subprogram must not appear in an EQUIVALENCE or COMMON statement within the subprogram.

A SUBROUTINE subprogram can redefine any of the actual arguments or use any of the dummy arguments specified to return results to the calling program. For example:

```
SUBROUTINE COMPUTE (A,B,ANS)
ANS=A+B
A=A+1.0
RETURN
END
```

In this simple subroutine, the calling statement provides numeric values for A and B and a variable name in which the value of ANS is returned. If the calling statement is:

```
CALL COMPUTE (AI,4.0,RET)
```

where the value of AI is 7.0, the value returned to the calling program in RET is 11.0, and AI is 5.0.

A SUBROUTINE subprogram cannot contain a FUNCTION, BLOCK DATA or other SUBROUTINE statement.

6.4.5 CALL Statement

A subroutine is invoked by means of a CALL statement having one of the following forms:

CALL name

or

CALL name(arg₁, ..., arg_n)

where name is the name of the subroutine referenced and the argument list of the subroutine call, where included, must agree in number, order and type with the arguments in the subroutine definition. The actual arguments specified in the subroutine call replace the dummy arguments during execution of the subroutine.

The arguments of a CALL statement can be expressions, array names or elements, scalar variables, alphanumeric strings, or subprogram names. The subroutine name cannot be referenced as a basic element in an expression in the calling program or subroutine itself.

A call to a subroutine may contain fewer or more actual arguments than the subroutine definition as long as:

- a. an unsupplied call argument is not a dummy array in the subroutine, and
- b. some indication of the number of real arguments is given to the subroutine so that no attempt is made to reference unsupplied arguments.

6.4.6 RETURN Statement

The RETURN statement is of the form:

RETURN

and causes an exit from the subprogram to the calling program, and is normally the last executable statement in a subprogram. Any number of RETURN statements can appear in either a FUNCTION or SUBROUTINE subprogram.

The RETURN statement must not occur in a main program.

An END statement should occur following each subprogram. However, if a subprogram contains an END statement and no RETURN statement, executing the END statement causes a return to the calling program.

6.4.7 Assembly Language Subprograms

Subprograms can be written in the MACRO-11 assembly language for use with PDP-11 FORTRAN IV programs.

All linkages to subroutines and external and intrinsic functions are performed by the standard subroutine calling convention. The assembly language equivalent of the statement:

```
CALL XXX(a1,a2,...,an)
```

is as follows:

```
.GLOBL XXX
JSR    %5,XXX
BR     $FNXXX
.WORD  a1
      .
      .
      .
.WORD  an
$FNXXX:
```

where XXX is the name of the subprogram called and a1,...,an represents the addresses of the subprogram arguments. The convention is to:

- a. Perform a JSR %5,XXX where XXX is the routine to be called.
- b. Perform a BR .+n where n is the number of bytes+2 in the parameter list.
- c. Specify the addresses of zero or more parameters in-line with the call.

The parameters can be referenced by means of indexed references using R5 as the index register; i.e., 2(R5) obtains the address of a1, 4(R5) obtains the address of a2, etc. Other ways of obtaining parameters include using (R5)+ references to obtain parameters in sequential order. In this case R5 must be advanced past the branch instruction to obtain all parameters so that when the RTS R5 instruction is encountered following the subroutine, R5 is

pointing to the proper location (following the parameter list).

When calling assembly language subroutines, it may be useful to allow a variable number of arguments to the subprogram. This can be easily done since the BR instruction has, in its low-order byte, a word offset to its target address. In the case of the parameter lists, the word offset equals the number of parameters. Thus, performing a:

```
MOV (R5)+,LOC
```

obtains the branch instruction. Clearing the high-order byte of LOC will leave the correct number of parameters in LOC.

```
.TITLE ADDIT
;
;SAMPLE ASSEMBLY ROUTINE TO ADD TOGETHER AN ARBITRARY
;NUMBER OF INTEGER ARGUMENTS SUPPLIED BY A FORTRAN
;FUNCTION CALL STATEMENT.
;
.MCALL .PARAM ;GET .PARAM MACRO FROM SYSTEM
;MACRO FILE
.PARAM ;SPECIFY REGISTER PARAMETERS ETC.
;
.GLOBL IADD
IADD: MOV (R5)+,R3 ;GET THE NUMBER OF PARAMETERS
BIC #177400,R3 ;CLEAR THE EXTRANEIOUS BITS
CLR R0 ;PRESET PARTIAL SUM TO ZERO
1$: DEC R3 ;DECREMENT PARAMETER COUNT
BMI 2$ ;QUIT WHEN ALL PARAMETERS ARE
;PROCESSED
ADD @(R5)+,R0 ;ADD AN ARGUMENT TO PARTIAL
;RESULT
BR 1$ ;LOOP UNTIL ALL ARGUMENTS ARE
;PROCESSED
;
;AT THIS POINT ALL OF THE PARAMETERS HAVE BEEN ADDED
;TOGETHER AND THE RESULT IS IN REGISTER 0.
;
2$: RTS R5 ;RETURN TO THE CALLER WITH
;THE RESULT IN R0
.END
```

Example calls to the above routine could be written as:

```
J=IADD(J,K,2,L)
L=2+IADD(1,2,L,K)
```

6.4.8 BLOCK DATA Subprogram

The BLOCK DATA subprogram is used to enter initial values for variables assigned to labeled COMMON blocks. The BLOCK DATA subprogram is of the form:

```
BLOCK DATA
.
.   (Specification statements)
.
END
```

The first statement must be a BLOCK DATA statement which does not allow either a line number or any arguments. No executable statements can appear in a BLOCK DATA subprogram. The entire BLOCK DATA subprogram can contain only EQUIVALENCE, DATA, DIMENSION, COMMON, IMPLICIT and type declaration statements. A complete set of specifications is given for an entire common block by a single BLOCK DATA subprogram, although a single BLOCK DATA subprogram can initialize any number of named common blocks. For example:

```
BLOCK DATA
COMMON/R/S,Y/C/Z,W,Y
DIMENSION Y(3)
COMPLEX Z
DATA Y/1E-1,2*3E2/W,Z/11.877E0,(-1.41421,1.41421)/
```

6.5 SYSTEM SUBROUTINES

Like the functions intrinsic to the FORTRAN system, there are subroutines in the FORTRAN Library which the user may call in the same manner as a user-written subroutine. These subroutines are:

PDUMP	performs core dumps of specified sections of core.
SETERR	allows the user to reset the maximum occurrence count for any class of error except 0.
SETFIL	overrides default values for FORTRAN logical device assignments.
EXIT	terminates the execution of a program and returns control to the Monitor (also invoked automatically by the error processing routines and STOP statement).
DATE	returns a 9-byte string containing the ASCII representation of the current date.
TIME	returns an 8-byte string containing the ASCII representation of the current time in hours, minutes and seconds.

SSWTCH	provides a means to sense the console switch settings.
RANDU	returns a single random number with a uniform distribution between 0 and 1.
TSTERR	returns an indication of whether an error of the specified class has occurred.
LINK, RETURN	handle overlays (see Chapter 7).
RUN	loads and executes another program (see Section 7.12).

6.5.1 PDUMP

The PDUMP subroutine causes specified portions of core to be dumped. Control returns to the calling program following execution of the dump. The call to PDUMP is performed as follows:

```
CALL PDUMP (L1, U1, F1, ..., Ln, Un, Fn)
```

where either L_i or U_i can be the lower limit (variable name) of the area to be dumped, with the remaining variable indicating the upper limit. Regardless of the specification order, storage is always dumped from the lower limit to and including the upper limit.

```
DIMENSION N(10)
      :
      CALL PDUMP (N(1), N(10), 3)
```

which causes the array N to be dumped on the error logging device as a string of two-word integers from N(1) to N(10). The error logging device is generally KB: or LP: (see Section 10.5 for a description of how to change the error logging device). The following line performs the same dump as that performed by the call above:

```
CALL PDUMP (N(10), N(1), 3)
```

The value of F_i is an integer constant or variable indicating the dump format as follows:

<u>Code</u>	<u>Description</u>	<u>Format</u>
Ø	1-word octal	808
1	1-word decimal	8I8
2	2-word octal	808
3	2-word decimal	8I8
4	real	4G15.7
5	double precision	3D22.14

Codes Ø and 1 should be used if the /ON (one-word integer) switch is set, and codes 2 and 3 should be used if the /ON switch is not set. Codes Ø and 1 print every word while codes 2 and 3 print every other word, since the alternate words are unused. No default dump mode is assumed; a code must be specified or a diagnostic is generated. Where several sections of core are to be dumped, each set of three values specifies a separate core area and dump format.

No spacing is performed between dumps of individual sections and no addresses are printed on the PDUMP output.

As an example, the output shown below was produced using the call:

```
CALL PDUMP (B(1),B(3),Ø,R(1),R(7),Ø,F(1),F(4),5,S(1),S(24),Ø)
```

where the program contains:

```
DOUBLE PRECISION F(4)
INTEGER B(3),R(7),S(24)
```

and the /ON (one-word integer) switch is used at compile time.

```

14Ø4      14      12746
  Ø        Ø        Ø        Ø      2Ø124      2Ø13Ø      17776
        Ø.      Ø.197215245Ø87Ø1D-3Ø      Ø.986Ø762431Ø639D-31
Ø.493Ø3812265972D-31
22Ø6Ø      3477Ø      17756      5ØØØ2      3Ø      21172      4Ø6      2Ø13Ø
  Ø        4467      12672      3224Ø        Ø        Ø      5Ø561      5574Ø
327Ø2      1      2ØØ24      4567      1412      4ØØ      4467      1264Ø
```

6.5.2 SETERR

An error count is maintained by the system for each error class. The SETERR subroutine allows the user to reset the maximum occurrence count for any class of run-time error except 0 or 8. (Following the maximum number of errors in a given class, control exits to the Monitor.) The call to SETERR is performed as follows:

```
CALL SETERR(CLASS,MAX)
```

The argument CLASS is an integer indicating the error class affected. If CLASS is not a valid number, no action is performed. The CLASS argument is shown in relation to its associated error messages in Appendix E. In brief, the error classes are:

<u>Class</u>	<u>Default MAX Value</u>	<u>Meaning</u>
0	-3	fatal errors; it is recommended that the maximum occurrence count (-3) not be changed
1	0	physical I/O errors
2	1	errors in FORMAT statements
3	3	arithmetic overflow or division by 0
4	4	incorrect arguments to library functions or subroutines
5	-1	arithmetic underflow errors
6	0	conversion errors
7	7	subscripting errors
8	1	errors involving incorrect linkages to OTS routines; the maximum occurrence count (1) cannot be changed, although no diagnostic is given

The argument MAX is an integer with the following significance:

```

MAX = >0, log message, increment error count, call EXIT
      if error count is ≥ MAX
      0, log messages and ignore error count for the
        specified error class
      -1, do not log messages and ignore error count
        for the specified error class
      -2, do not log messages, exit to Monitor which
        prints an I351 message, followed by a $
        character
      -3, immediate exit to Monitor with a message of
        the form:
        F030 00C00N
        where C and N are the class and message
        number (see Appendix E) of the error causing
        the exit, followed by a $ character

```

6.5.3 SETFIL

The SETFIL subroutine overrides the default value for a logical device assignment at run time. The SETFIL call must occur before the logical unit is opened for I/O operations (by READ or WRITE). The new logical device assignment remains in effect until the end of the current program or until the file is closed by ENDFILE and a new device assignment is made. The call to SETFIL is performed as follows:

CALL SETFIL (N,FILE,IERR,DEV,NU,UIC,PC,CS,LREC,NREC)

SETFIL accepts a variable number of arguments; trailing arguments can be omitted. The complete list of arguments to this call are described below:

<u>Argument</u>	<u>Significance</u>
N	logical unit number expressed as an integer value.
FILE	character string containing the filename and extension specification, of the form: file.ext where file can be up to six alphanumeric characters, the first of which is alphabetic. Optionally, .ext can be specified to indicate an extension. The entire file specification is an ASCII string which can be expressed as a Hollerith string or enclosed in quotes. For example: 'MAT.TMP' 8HOB2.DAT See also the discussion of string argument storage formats in Section 6.5.10.
IERR	integer variable into which the error returns from this routine or from I/O operations on logical unit N are placed. IERR is set to -1 if CS≠1 or 2, or to 2 if CS=1 and LREC and NREC are not specified.
DEV	2-or 3-character device specification expressed as a Hollerith string or enclosed in quotes. For example: 'DT' 2HLP
NU	unit number of device specified, if appropriate. This is an integer constant or variable (e.g., 1 if device is DEctape unit 1, or 0 if line printer or other single unit device).
UIC	user identification code (UIC) expressed as an integer value. A UIC specification of 0 indicates the current UIC and is the usual specification. A simple way to prepare the UIC entry is as follows (assume UIC desired is 120,150):

```

      INTEGER UIC
      LOGICAL*1 UICB(2)
      EQUIVALENCE (UICB(1),UIC)
      UICB(1) = "15Ø
      UICB(2) = "12Ø

```

```

      :

```

```

      CALL SETFIL(2,'DATA.TMP',IERR,'DT',2,UIC)

```

which specifies the file DT2:DATA.TMP [12Ø,15Ø]
for access on logical device 2.

PC	protection code specified as an integer value which is the decimal equivalent of the desired octal protection code (protection codes are defined with octal values in the DOS Monitor manual). The default FORTRAN protection code is 233 (octal).
CS	integer value used to indicate one of the following: CS = 2, allocate a contiguous file for random I/O. The DEFINE FILE statement sets the record length and number of records in the file. CS = 1, allocate a contiguous file for formatted or unformatted I/O. The arguments LREC and NREC are used to set the record length and number of records in the file.
LREC	logical record length in words expressed as an integer value; required if CS = 1, otherwise ignored.
NREC	number of records to allocate expressed as an integer value; required if CS = 1, otherwise ignored.

The discussion of file formats in Section 10.2 should be consulted as a guide to choosing LREC and NREC.

When SETFIL is used in an overlay system, there is a special restriction on the variable IERR. In the statement:

```

      CALL SETFIL(7,'FILE.DAT',IERR,'DK',Ø)

```

the address of variable IERR is saved and is used by the I/O system for reporting certain kinds of I/O error conditions. This variable name is saved by SETFIL in the device table, \$DEVTB. Any later I/O error (during a READ or WRITE) attempts to set IERR. Therefore, IERR must be in the local storage of the resident section or in a resident common block. If it is not, the FORTRAN I/O system may destroy an unpredictable location in memory when attempting to store into IERR.

Further discussion of the use of SETFIL and of the FORTRAN I/O system can be found in Chapter 10. Two examples of the use of SETFIL are given below. These examples assume the default logical unit assignments.

Example 1: Make logical unit 5 the card reader and logical unit 6 the line printer.

```
CALL SETFIL(5,'A',IERR,'CR')
CALL SETFIL(6,'A',IERR,'LP')
```

Note that the parameters to SETFIL are positional. The CR: and LP: are not file structured and the filename argument is superfluous; some filename argument must be included, however.

Example 2: Allocate a file for direct-access I/O on unit 1 under the current UIC.

```
CALL SETFIL(1,'FILE1.XYZ',IERR,'SY',0,0,"100,2)
DEFINE FILE 1(20,200,U,INDX1)
```

The file FILE1.XYZ is created on the system disk with protection code 100 (read, write, execute access; the file cannot be deleted until its protection code is changed).

6.5.4 RANDU, RAN

The random number generator can be called as a subroutine or as an intrinsic function, RAN. The subroutine call is performed as follows:

```
CALL RANDU (I1,I2,X)
```

where I1 and I2 are previously defined integer variables and X is the real variable name in which is returned a random number between 0 and 1. I1 and I2 should be initially set to 0. I1 and I2 are updated to a new generator base during each call. Resetting I1 and I2 to 0 repeats the random number sequence. The values of I1 and I2 have a special form; only 0 or saved values of I1 and I2 should be stored in these variables.

Use of the random number subroutine is similar to the use of the RAN function where:

```
X=RAN(I1,I2)
```

is the functional form of the random number generator.

6.5.5 EXIT

A call to the EXIT subroutine, in the form:

```
CALL EXIT
```

is equivalent to the END statement and causes a return to the Monitor.

6.5.6 DATE

The DATE subroutine can be used in a FORTRAN program to obtain the current date as set within the system. The DATE subroutine is called as follows:

```
CALL DATE(array)
```

where array is a predefined array able to contain a 9-byte string. The array specification in the call may be expressed as the array name alone:

```
CALL DATE(A)
```

in which the first three elements of the real array A are used to hold the date string, or as:

```
CALL DATE(A(i))
```

which causes the 9-byte string to begin at the ith element of the array A.

The date is returned as a 9-byte (9-character) string in the form:

```
dd-mm-yy
```

where: dd is the two-digit date
mm is the three-letter month specification
yy is the last two digits of the year

For example:

```
15-NOV-72
```

In the case where the array is a real array, 4-1/2 words are used to contain the date string with the remaining array storage being untouched. Therefore, the date string is stored in the first 9 bytes in the elements A(i), A(i+1), and A(i+2). The last 3 bytes of A(i+2) are untouched and should be made blank by the user if he intends to print the date with a 3A4 format.

6.5.7 TIME

The TIME subroutine allows the user to access the current system time or perform time conversions within a FORTRAN program. The TIME subroutine has three different calls:

```
CALL TIME(A)
CALL TIME(I1,I2)
and
CALL TIME(A,I1,I2)
```

The first two calls read the current system time; the third is used to perform time conversions.

The first type of TIME call returns the time as an 8-byte (8-character) ASCII string of the form:

hh:mm:ss

where: hh is the two-digit hour indication
mm is the two-digit minute indication
ss is the two-digit second indication

For example:

10:45:23

A 24-hour clock is used. This quantity is typically output with a 2A4 format specification.

The second type of TIME call returns the time in clock ticks (1/60 of a second for 60 Hz systems, 1/50 of a second for 50Hz) since midnight in the integer variables I1 and I2. I1 contains the high-order 15-bits of the number and I2 the low-order 15-bits. Elapsed time calculations can be performed as follows:

```

      .
      .
      .
CALL TIME(I1,I2)
      .
      .
CALL TIME(I3,I4)
C    COMPUTE ELAPSED TIME IN TICKS; FIRST CONVERT TICKS STORED
C    IN I1 AND I2 INTO A SINGLE, FLOATING-POINT VALUE.
      START = I1*32768.0+I2
      FINISH = I3*32768.0+I4
C    NOTE THAT MULTIPLICATION BY REAL CONSTANT CAUSES ENTIRE
C    CALCULATION TO BE PERFORMED IN REAL MODE.  ELAPSED TIME IS
C    NOT CALCULATED IN INTEGER MODE DUE TO THE POSSIBILITY OF
C    INTEGER OVERFLOW.
      ELAPSE = FINISH-START
C    CONVERT TO FIXED-POINT
      I1 = ELAPSE/32768.0
      I2 = AMOD(ELAPSE,32768.0)
      WRITE(6,210) I1,I2
210  FORMAT(206)
      .

```

The above example prints a time indication as follows:

```

      0  2630

```

which provides an octal representation of the number of clock ticks.

The third type of TIME call accepts I1 and I2 as integer values and returns the clock time as represented by I1 and I2 in an 8-byte (8-character) ASCII string in A. To continue the previous example:

```

      CALL TIME(A(1),I1,I2)
C    SCALE ELAPSED TIME TO SECONDS
      ELAPSE = ELAPSE/60.0
      WRITE(6,215) ELAPSE,A(1),A(2)
215  FORMAT('ELAPSED TIME = ',F5.1,' SECONDS'/
1' AS CONVERTED BY "TIME" : ',2A4/)

```

which causes the following to be output:

```

ELAPSED TIME = 23.9 SECONDS
AS CONVERTED BY "TIME" : 00:00:23

```

6.5.8 SSWTCH

The sense switch (SSWTCH) subroutine is available for use in FORTRAN programs to test specified bits of the console switch register. The call is performed as follows:

```
CALL SSWTCH (I,J)
```

where:

I is the bit of the console switch register to be tested.
I must be in the range $0 \leq I \leq 15$. If I is out of range, no value is returned in J and no diagnostic is printed

J contains a 1 if bit I is 1 (UP)
2 if bit I is 0 (DOWN)

6.5.9 TSTERR

The TSTERR subroutine allows the user program to monitor the types of errors detected during program execution. The call is of the form:

```
CALL TSTERR(I,J)
```

where I is supplied as the error class number (between 0 and 8) and the value of J is returned as:

J = 1 if an error of class I has occurred
J = 2 if an error of class I has not occurred

The sequence:

```
      :  
      :  
      CALL TSTERR(3,J)  
      GO TO(10,20),J  
20  CONTINUE  
      :  
      :
```

transfers control to statement 10 if a class 3 error (arithmetic overflow) has occurred. See Section 6.5.2 and Appendix E for a discussion of error classes and messages.

The TSTERR routine also resets to 0 the error flag for that error class (but not the error count used by SETERR). For example:

```
      :  
      CALL TSTERR(I,J)  
      CALL TSTERR(I,J)  
      :
```

The second call is guaranteed to return J = 2. The TSTERR subroutine is independent of the SETERR subroutine; neither directly influences the other except that SETERR can cause execution to terminate.

6.5.10 Character String Arguments to System Subroutines

String arguments to system subroutines must be stored in adjacent bytes. String values stored in integer arrays (see Section 12.2 and Appendix B) are not stored in adjacent bytes unless the /ON switch is used in compiling the program; such character strings are unacceptable as arguments to system subroutines.

The string values returned by the DATE and TIME subroutines are stored in adjacent bytes independent of the use of the /ON switch.

Object time format specifications (see Section 8.16) must also be stored in adjacent bytes.

String arguments to the SETFIL, LINK and RUN subroutines must be stored in adjacent bytes and have a terminating zero byte. The FORTRAN system automatically appends a zero byte to character string arguments specified as Hollerith constants. For example:

```
      CALL SETFIL(2,'MAT.TMP')  
      CALL LINK(8HOVLI.LDA)
```

If a character string value is stored as a variable or an array, the FORTRAN program must append a zero byte to the string.

CHAPTER 7

PROGRAM OVERLAY FACILITY

7.1 INTRODUCTION TO THE OVERLAY FACILITY

The PDP-11 DOS program overlay facility enables the user to have virtually unlimited core space for his program. A program using the overlay facility can be much larger than would normally fit in the available core space, since portions of the program, called overlays, reside on the backup storage (that is, disk, DECTape, or magtape) and are brought into core as needed.

Overlays are useful when handling large FORTRAN programs and can also be used with assembly language programs.

The run-time supervisor (which is linked to the resident section of the user program) of the overlay facility supplied with the PDP-11 FORTRAN system is the subroutine LINK in the FORTRAN Library version V020A.

The resident portion of the run-time supervisor requires approximately 150₁₀ words. However, the Monitor requires an additional 768₁₀ words for buffers and approximately 150₁₀ words for the overlay stack at the time the overlay is loaded.

The overlay system which the user constructs from his completed program is composed of a core resident section, a core resident overlay area, and overlay files stored on backup storage.

For example, a FORTRAN system could contain a resident section (including the run-time supervisor) and maintain an overlay area into which any number of overlay files could be moved as required.

Link-11 (V011A or later version) is used to link the core resident section of the user program to the Library routines required (including the overlay supervisor). Link-11 is also used to create the various overlay files to be used with the resident section.

Overlays communicate through blank or labelled COMMON in the resident area or through the contents of general registers R0

through R4 and the stack. Overlay files can use any core resident routines (user generated or acquired from the FORTRAN Library).

The user is advised to organize the overlay strategy for his program prior to program development to provide for COMMON usage, format conversions, and core residency requirements.

7.2 COMMUNICATION AMONG RESIDENT AND OVERLAY ROUTINES

Overlays and the resident section communicate through blank and labelled COMMON areas (.CSECT's for assembly language programs) in the core resident area. Data can also be passed between overlays (from assembly language programs) in the general registers R0 through R4 and the stack.

7.3 FORMAT CONVERSIONS AND I/O ROUTINES

Any format conversions and I/O control routines not needed in the resident section but required by overlay sections must be forcibly loaded into the resident section. This can be done by declaring the appropriate globals in an assembly language routine or inserting dummy FORMAT and Input/Output statements in the resident main program for all those routines needed in the overlays and not required in the resident section.

For example, assume I and L format conversions are needed for READ, and I and E format conversions are needed for WRITE. An assembly language routine such as the following could be written:

```
.TITLE DUMMY
.GLOBL $LCI,$ICI,$ICO,$DCO
.END
```

where \$LCI performs the L conversions for READ, \$ICI performs the I conversions for READ, \$ICO performs the I conversions for WRITE, and \$DCO performs the E conversion for WRITE.

Alternatively, dummy FORMAT and Input/Output statements can be supplied in the resident main program to force linking of these routines. (If this is done, a message is printed at compile time indicating that there is non-executable code in the program.) For example:

```

LOGICAL L
.
.
.
GO TO 1000
READ (6,1000) I,L
1000 FORMAT (I1,L1)
WRITE (6,101) I,E
101 FORMAT (I1,E6.0)
10000 CONTINUE
.
.
.

```

Including global references in an assembly language routine causes only the four format conversion packages to be linked to the resident program section. Inserting dummy FORMAT and Input/Output statements causes the resident to carry the overhead of the four format conversion packages plus the FORTRAN READ/WRITE processor, FORMAT scanner, and associated routines.

In addition to the format conversion packages and basic I/O control routines, the control routines for unformatted and direct-access I/O are separately loadable modules. Where any I/O is performed in the resident section, the formatted I/O control routines are loaded in the resident section; the unformatted and/or direct-access routines are loaded only if the resident contains statements requiring their presence. If an overlay performs unformatted or direct-access I/O, it is necessary to force the loading of the appropriate module(s) into the resident routines. The routines can be force loaded by means of an assembly language routine containing the global symbols:

```

unformatted I/O:      $IOUD
direct-access I/O:    $IORD

```

To force loading of these routines by means of dummy I/O statements, the following statements could be included in the resident section:

```

C  FORCE LOADING OF UNFORMATTED I/O
GO TO 110
1000 READ(1) I
110 CONTINUE

```

or:

```

C  FORCE LOADING OF DIRECT-ACCESS I/O
GO TO 110
1000 READ(1'1) I
110 CONTINUE

```

Two other possibilities are either to perform all I/O in the resident program only, or to perform all I/O in the overlay section. If there is no I/O in the resident section, each overlay includes only those modules needed to satisfy its own I/O requirements.

If those format conversions and Input/Output routines which are needed in the overlays and not required in the resident section are not forcibly loaded into the resident section, the FORTRAN system causes the linking of dummy routines. Global requests in the overlay files are then linked to the resident dummy routines and at execution time results in one of the fatal error messages:

```
FORT008000 LINKAGE ERROR (MISSING FORMAT CONVERSION ROUTINE)
FORT008001 LINKAGE ERROR (UNFORMATTED I/O)
FORT008002 LINKAGE ERROR (DIRECT ACCESS I/O)
```

If it is essential to minimize the amount of memory used by the resident section, the technique of forced loading of modules by means of an assembly language routine is recommended. The assembly language routine does not force all routines in the I/O package into the resident section, but rather causes the loading of some modules which would otherwise be blocked. The resulting resident section may be smaller than that produced by the inclusion of the dummy FORTRAN statements shown above.

Table 7-1 is useful in building overlay systems. If any module not needed by the resident is required in an overlay, then the corresponding global must be declared in the resident section.

NOTE

See Section 6.5.3 describing the SETFIL routine for a restriction present when SETFIL is used with an overlay system. Section 9.4.1 describes a similar situation with respect to DEFINE FILE.

7.4 LINK, THE RUN-TIME OVERLAY SUPERVISOR

The subroutine LINK is the run-time overlay supervisor. It has three entry points: LINK, RETURN and RUN (for RUN, see Section 7.12).

The subroutine LINK performs the following functions for the user when entered at LINK:

- a. initializes the traceback origin to the new overlay file to allow FORTRAN error tracking via the traceback feature. (See Section 12.4.)

Table 7-1

Format Conversion Packages and I/O Routines

Globals in Package	Function Performed	Length of Package in Decimal Words*
\$DCO } \$ECO } \$FCO } \$GCO }	Output conversions D,E,F,G	469
\$ICO } \$OCO }	Output conversions, I,O	93
\$LCO	Output conversions L	31
\$DCI } \$RCI }	Input conversions D,E,F,G	384
\$ICI } \$OCI }	Input conversions I,O	85
\$LCI	Input conversions L	31
\$IOUD	Unformatted I/O control routine	234
\$IORD	Direct-access I/O control routine	273

- b. if called from the resident section saves general registers R0 through R5 and saves the return address to the resident section for use with the next CALL RETURN.
- c. brings the overlay file (designated by the argument to CALL LINK) into core and moves the stack below the new overlay file.

The subroutine LINK performs the following functions for the user when entered at RETURN:

- a. restores general registers R0 through R5.
- b. transfers control to the statement following the last CALL LINK executed from the resident section.

The LINK subroutine contains the global declaration:

```
.GLOBL $OTSV
```

* Includes certain associated modules.

where \$OTSV is the pointer to the FORTRAN impure area. This forces the impure area into the core resident section so that user I/O can be continued across overlays.

7.5 CALLING AN OVERLAY FILE

The resident section and overlay files are entered into the system as load modules. An overlay is requested by a

CALL LINK ('A')

statement, where A is the file specification of the overlay file to be called into core. All overlays are entered through their main program, not through any subroutines.

The argument of the overlay call is enclosed in single quotes since the Linker relies on a terminating null character inserted by the FORTRAN Compiler at the end of such a string. (See the discussion of string argument storage formats in Section 6.5.10.) The maximum length of the argument is 25 characters (stored one character per byte).

The call can be written as follows:

CALL LINK ('dev:file.ext[uic]')

Using the DOS default conventions, the call is normally expressed as:

CALL LINK('file')

with the system disk the assumed device; .LDA the assumed extension for a load module; and the current UIC assumed.

Whenever the statement:

CALL RETURN

is encountered, control transfers to the instruction following the last CALL LINK executed from the resident section.

The first CALL LINK statement is issued from somewhere in the core resident section of the user program, bringing into core the first overlay file. Subsequent CALL LINK statements can be issued from the main programs in the various overlays. Each CALL LINK brings into core the specified overlay file.

CALL LINK can be issued from anywhere in the core resident section (main program or subroutines). CALL LINK and CALL RETURN can be issued only from the main program in an overlay and not from a subroutine within an overlay.

There are four allowable control paths between resident and overlay load modules:

- a. resident to overlay via CALL LINK
- b. overlay to overlay via CALL LINK
- c. overlay to resident via CALL RETURN
- d. overlay utilizing resident subroutines.

Figure 7-1 illustrates these transfer paths:

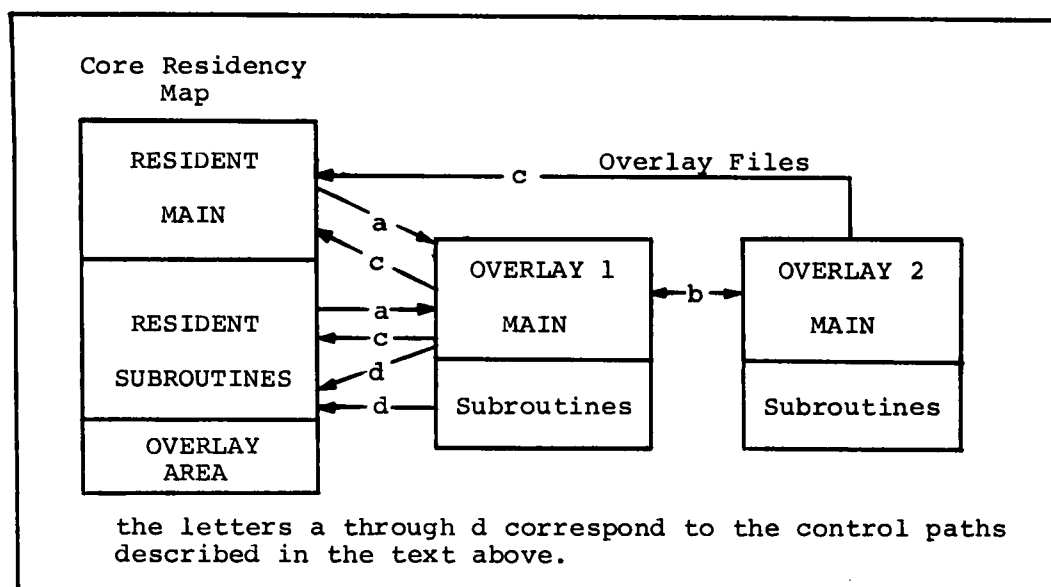


Figure 7-1
Transfer Paths Between Resident and Overlay Programs

7.5.1 Overlay Transfer Paths

CALL LINK may be issued from anywhere in the resident section. However, in an overlay, CALL LINK and CALL RETURN can be issued only from the overlay main program.

Resident subroutines can be called from anywhere in an overlay file (main program or subroutine). However, there is a restriction that an overlay must not call any resident routine which contains (or calls another routine which contains) a CALL LINK.

Violation of these control path restrictions causes the overlay system to be corrupted, resulting in stack overflow or incorrect subroutine returns.

7.5.2 Search for Overlay Files

Depending upon the format of the overlay file specification, a CALL LINK statement searches for the overlay file under several possible, alternate, file specifications.

Where only the overlay filename was specified, for example:

```
CALL LINK('FILE')
```

the search for the overlay file proceeds as follows:

- (1.) FILE.LDA, current UIC
- (2.) FILE.LDA, 1,1 UIC
- (3.) FILE, no extension, current UIC
- (4.) FILE, no extension, 1,1 UIC

Where a filename and extension are specified for the overlay file, for example:

```
CALL LINK('FILE.EXT')
```

the search for the overlay file proceeds as follows:

- (1.) FILE.EXT, current UIC
- (2.) FILE.EXT, 1,1 UIC

Where a UIC is specified and a file extension is not, for example:

```
CALL LINK('FILE[X,X]')
```

the search for the overlay file proceeds as follows:

- (1.) FILE.LDA, X,X UIC
- (2.) FILE, no extension, X,X UIC

Where a complete file specification is given, only one attempt is made to find the file. For example:

```
CALL LINK('FILE.EXT[X,X]')
```

is searched under:

- (1.) FILE.EXT, X,X UIC

If the search has failed, the DOS Monitor prints the message:

```
F012 XXXXXX
```

where the additional information word contains the request address (which is within the run-time supervisor).

7.6 OPERATING PROCEDURES

In order to create an overlay system, the FORTRAN main program and various overlay files are separately compiled and linked using Link-11 (V011A). The /OV switch is placed in a command string to the Linker containing the FORTRAN resident module. The argument to the /OV switch indicates the number of overlay files. One command string is passed to the Linker for each overlay file. The argument to the /OV switch, then, indicates the number of command strings for the Linker to expect. For example:

```
$RU LINK  
LINK-11 V011A  
#RES,LP:<RES/OV:2,FTNLIB/L/E  
#OVL1,LP:<OVL1,FTNLIB/L/E  
#OVL2,LP:<OVL2,FTNLIB/L/E
```

When the Linker receives the first command string, the /OV switch causes it to enter overlay mode. The next two command strings are treated as overlay files to be linked into an overlay system with the resident section RES.LDA.

7.6.1 Linker Command String Formats

The format of the Linker command string which defines the resident section is as follows:

dev:RES,dev:MAP,dev:ST<dev:RES/OV:n,...subroutines,FTNLIB/L/E

where:

dev:RES	is the file specification for the user's core resident section load module.
dev:MAP	is the load map of the resident program, which is an optional output file.
dev:ST	is the global symbol table of the resident program load module, which is an optional output file.
dev:RES	is the file specification for the user's core resident main program
/OV:n	specifies that n overlay files are to be defined in the following n Linker command strings.
... subroutines	indicates the user subroutines to be linked to the main program.
FTNLIB/L	the FORTRAN Library contains the overlay facility supervisor and other Library routines which must be linked to the user's resident section. /L switch specifies that the file FTNLIB is a Library file.
/E	indicates the end of the command string to the Linker. This is followed by the RETURN key.

Following the command string describing the resident module are the command strings describing the overlay files. The format of these command strings is as follows:

```
dev:OVER,dev:MAP<dev:OVER1,...subroutines,FTNLIB/L/E
```

where:

dev:OVER	is the file specification for the overlay load module. This specification must correspond exactly to the one used as the argument to CALL LINK when this overlay is wanted.
dev:MAP	is the load map of the overlay, which is an optional output file.
dev:OVER1	is the file specification for the overlay main program.
...subroutines	indicates the user subroutines to be linked to this particular overlay, if any.
FTNLIB/L	the FORTRAN Library must be linked to each overlay. The /L switch specifies the file FTNLIB as a library file.
/E	indicates the end of the command string to the Linker. This is followed by the RETURN key.

Notice that in the command strings describing the resident and overlay files the FORTRAN Library, FTNLIB, is linked to each load module.

Since the default device for load modules is always the system device and the default extension is .LDA, the load module file specification can be simplified for both resident and overlay file creation as follows:

```
RES,dev:MAP,dev:ST<dev:RES/OV:n,...subroutines,FTNLIB/L/E  
OVER,dev:MAP<dev:OVER1,...subroutines,FTNLIB/L/E
```

The structure of the program overlay facility allows modifications to be made to overlay load modules without relinking the resident and other overlay load modules, (see Section 7.6.2).

7.6.2 Operations Performed by the Linker

The top of the overlay files is automatically set to the location directly below the bottom of the resident section. The /T:n Linker switch is no longer necessary, but can be used to change the default top linkage, if desired. The top of the overlay should never be set to a location above the bottom of the resident section.

In the example shown in Section 7.6 the resident symbol table is maintained in core memory and is discarded after the overlay system is linked. If the user can foresee future additions or changes in the overlay files, he should output the resident symbol table so that future changes can be made without relinking the entire overlay system. For example:

```
$RU LINK  
LINK-11 V011A  
#RES,LP:,SY:RES<RES/OV:2,FTNLIB/L/E  
#OVL1,LP:<OVL1,FTNLIB/L/E  
#OVL2,LP:<OVL2,FTNLIB/L/E
```

The first command string to the Linker causes the resident section symbol table (SY:RES.STB) to be maintained on the system disk. This symbol table file contains the global symbols of the resident section and the bottom address of the resident.

At a later date a new overlay file can be added to the present system by means of the following command sequence:

```
$RU LINK  
LINK-11 V011A  
#OVL3,LP:<RES.STB,OVL3,FTNLIB/L/E
```

or, to relink a changed version of OVL2, the following command sequence is used:

```
$RU LINK  
LINK-11 V011A  
#OVL2,LP:<RES.STB,OV2NEW,FTNLIB/L/E
```

Notice that in the two examples above, the symbol table of the resident section is the first input file specification for each overlay file and the FORTRAN Library, FTNLIB, is linked to each overlay.

In the last example, the output filename cannot be altered since it is referenced in a call statement as OVL2 from either the resident and/or OVL1.

7.7 ERROR PROCEDURES AND MESSAGES

Error handling during the creation of the overlay system is performed as for Link-11. At run time there are three DOS fatal errors which are concerned with the overlay facility. These are as follows:

<u>Error Code</u>	<u>Explanation</u>
F275 XXXXXX	Argument to the LINK subroutine is syntactically incorrect or too long.
F276 XXXXXX	Transfer address of the overlay file was not specified. For FORTRAN overlays, no main program was included in the overlay file.
F277 XXXXXX	Overlay file could not be brought into core because it would overlay the resident section. This is the only error condition which generates the F277 code.

XXXXXX is the additional information word which contains the address in the user code following the offending CALL LINK.

After an F275 error message (where the additional information is the address following the offending CALL LINK), identify the offending CALL LINK and search for one of the following error conditions in the argument of CALL LINK:

- a. syntactically incorrect file specification,
- b. multiple file specifications,
- c. illegal switch specification, or
- d. file specification longer than 25 characters.

When the error is isolated, correct it. Reassemble or recompile the object module in which the error occurred. If the error occurred in the resident section, relink the entire overlay system. If the error occurred in an overlay file, then relink only that particular overlay file with the aid of the resident section symbol table and the FORTRAN Library.

After an F276 error message, determine the offending CALL LINK as for F275. If the overlay file is a FORTRAN program, the error was probably caused by not including a main program in the overlay file. If the overlay file was composed of assembly language modules, then the error was caused by not specifying a transfer address. Follow relinking instructions described under F275.

After an F277 error message, determine the offending CALL LINK as for F275. This error results when the overlay would have overlaid part of the resident section. To correct the error condition, the offending overlay must be relinked with a top of (low limit of the resident-2). See relinking instructions under F275. (This error does not occur if the correct symbol table (.STB) file is being

used since the symbol table file includes a pointer to the bottom of the resident section.)

7.8 ASSEMBLY LANGUAGE OVERLAYS

The creation of the overlay system (resident and overlay load modules) is the same for assembly language programs as for FORTRAN programs.

The assembly language calling sequence for the LINK subroutine is as follows (the standard FORTRAN calling sequence):

```
      R5=%5
      JSR R5, LINK
      BR .+4                      ;RETURN ADDRESS
      .WORD NAME                  ;POINTER TO ARGUMENT
      .
      .
NAME:  .ASCII /DEV:FILE/          ;ARGUMENT
      .BYTE 0                    ;ASCII CHARACTER STRING OF OVERLAY
      .EVEN                      ;LOAD MODULE FILE SPECIFICATION
                                   ;TERMINATING NULL BYTE
```

The assembly language calling sequence to return to the resident program is as follows (the standard FORTRAN calling sequence):

```
      JSR R5, RETURN
      BR .+2
      .
      .
      .
```

7.8.1 Global Declarations

In all assembly language programs that call LINK or RETURN, the symbols LINK and RETURN must be declared as globals.

```
      .GLOBL LINK, RETURN        ;EXTERNAL REFERENCES
```

7.8.2 Stack Usage

Assembly language resident code should not reset the stack at execution time to be at a lower address than set by the Monitor at loading time. The Monitor considers the stack as starting at the address set at loading time and ending at the contents of register 6 and uses this information to first decide if the stack must be moved and then to move it below an incoming overlay.

The stack cannot be addressed absolutely (e.g., pointers to the stack cannot be stored on the stack itself).

Exit from an overlay should be made only through the main program of that overlay, otherwise the return addresses from any nested subroutines are lost on the stack. (See Section 7.5.1.)

While executing FORTRAN programs, the stack is only used to store the return addresses from nested subroutines. (The latest entry on the stack is the old contents of register 5 and the return address to the last subroutine call is in register 5.)

With these restrictions in mind, the stack can be used by the assembly language program.

7.8.3 Register Usage

Before exiting from the resident program to an overlay, general registers R0 through R5 are saved by the LINK subroutine. On re-entering the resident program from an overlay, registers R0 through R5 are restored by the RETURN subroutine.

The register contents are not disturbed by LINK; therefore, arguments can be passed between overlays through the registers R0 - R4.

With these points in mind, the general registers can be used by the assembly language program.

7.8.4 Common Communication

Common communication between a FORTRAN main program and an assembly language subroutine is shown below. In the example, the variable I of blank common is input through the keyboard. First the two low-order bits then the five low-order bits of I are cleared by the mask in variable M of labelled common through the assembly language subroutine MASK.

Note that two words are allocated for integers in the assembly language subroutine MASK in order to be compatible with FORTRAN storage allocation.

The following is the FORTRAN main program:

```

COMMON I,J
COMMON /X/L,M
5  WRITE(6,1)
1  FORMAT(1H,7H INPUT )
C  INPUT VARIABLE I OF BLANK COMMON
C  FROM KEYBOARD

  READ(6,2) I
2  FORMAT(I5)

C  PREPARE THE MASK FOR
C  THE TWO LOW ORDER BITS
  M=3
  CALL MASK
  WRITE(6,3) I
3  FORMAT(1H ,2HI=,I5)
C  PREPARE THE MASK FOR BITS
C  0 THRU 4 (31(10) = 37(8)).
  M=31
  CALL MASK
  WRITE(6,4) J
4  FORMAT(1H ,2HJ=,I5)
  GO TO 5
END

```

The assembly language subroutine MASK follows:

```

      .TITLE MASK
      .GLOBL MASK
;CLEAR THE 1ST POSITION OF BLANK
;COMMON BY THE MASK OF THE 2ND POSITION
;OF NAMED COMMON, X.
MASK:
      R5=%5
      .CSECT   .$$$$.
      S=.%0
      J=.%4
      .=%10
      .CSECT X
      A=.%0
      B=.%4
      .=%10
      .CSECT
      BIC      B,S
      RTS      R5
      .END

```

7.9 PROGRAMMING NOTES

The DOS Monitor BEGIN command cannot be used to restart FORTRAN programs.

The DOS Monitor ASSIGN command is effective while using overlays. For example, where overlay files are specified (the argument of the LINK subroutine) to be obtained from the disk at execution time and those overlay files are moved onto DECTape unit 1, the command:

```
$AS DT1:,OVR
```

given immediately prior to running the overlay system load modules, causes all overlay files to be obtained from DECTape unit 1. Where the overlay files were originally specified as being on various devices, an assignment made to a particular device causes all overlay files to be searched on that device.

At present in Link-11 v011A, you can reinitialize common in the resident from an overlay via BLOCK DATA subprograms in FORTRAN or via a .WORD directive in assembly language. However, the use of this feature is discouraged because in future versions core image (contiguous) file overlays will not accommodate this feature.

When an A003 message appears during program execution, it may be because the link subroutine argument was not enclosed within quotes.

When an F010 message appears during a linking session, it could be because no device was specified for the third output field when LP: was specified as the second output field.

When using the DEFINE FILE or SETFIL routines in an overlay environment, the associated variable and error variable, respectively, must be stored in the resident section of the overlay system. See Sections 9.4.1 and 6.5.3 for details.

7.10 A SUMMARY OF THE MAJOR FEATURES OF THE OVERLAY FACILITY

An overlay system is composed of a resident section, a core resident overlay area and the overlay files. The user can specify the object modules to be included in the resident section or the overlay files at overlay build time. The resident section and overlay files are load modules. Overlays can use resident routines.

An overlay is requested via CALL LINK ('A'), where A is the file specification of the overlay which must contain a FORTRAN main program and, optionally, subroutines. CALL LINK ('A') can be issued from the resident section or from an overlay main program. CALL RETURN can be issued only from an overlay main program and returns control to the resident section.

Overlays communicate through blank and/or labelled COMMON (.CSECT's for the assembly language programmer) in the resident area. I/O can continue across overlays. Overlays can be modified without relinking the resident and other overlay load modules. Overlays are brought into core at each call.

7.11 PROGRAMMING EXAMPLE OF OVERLAY SYSTEM CREATION

The following pages contain an example of an overlay system which uses all the features of the PDP-11 overlay facility. A main resident program and resident subroutines are shown along with four short overlays.

Following the listing of the program source code is the disk directory before linking the overlay system, the command strings to the Linker, and the disk directory after linking the overlay system. Output from the run of the program is also shown.

The following is the source of the resident section, including the main program and subroutine.

```

C      THIS PROGRAM ILLUSTRATES THE USE OF OVERLAYS. IT CALCULATES
C      THE SQUARE ROOT OF SINE OF ALPHA. THE ANGLE ALPHA
C      IS INPUT IN OVERLAY 1. THE SINE OF ALPHA IS CALCULATED
C      IN OVERLAY 2. THE SQUARE ROOT OF SINE ALPHA IS FOUND IN OVERLAY
C      THE RESULTS ARE PRINTED IN OVERLAY 4.
C      LABELLED COMMON AREA FOR THE RESIDENT SECTION
C      AND FOR THE OVERLAYS.
0001  COMMON /X/A
C      BLANK COMMON AREA FOR THE RESIDENT SECTION
C      AND FOR THE OVERLAYS.
0002  COMMON B,C
0003  INTEGER BUFF1(10),BUFF2(10),BUFF3(10),BUFF4(10)
0004  DATA BUFF1/'IRE','ISI','IDE','INT',6*' '/
0005  DATA BUFF2/'ILA','IBE','IL2',7*' '/
0006  DATA BUFF3/'ILA','IBE','IL3',7*' '/
0007  DATA BUFF4/'ILA','IBE','IL4',7*' '/
C      CALL THE RESIDENT SUBROUTINE "PRINT" TO PRINT "RESIDENT"
C      TO INDICATE THAT EXECUTION STARTED IN THE RESIDENT SECTION.
0008  1  CALL PRINT(BUFF1)
C      CALL THE RESIDENT SUBROUTINE "LINK" TO BRING THE FIRST OVERLAY
C      INTO CORE FROM DISK. IN THE FIRST OVERLAY THE ANGLE ALPHA
C      IS INPUT FROM THE KEYBOARD WITH A FORMAT OF F8.2.
0009  CALL LINK('OVL1')
C      RETURNED TO RESIDENT AT LABEL 2 FROM OVERLAY 1 VIA
C      A CALL RETURN.
0010  2  CALL PRINT(BUFF2)
C      CALL THE RESIDENT SUBROUTINE "LINK" TO BRING INTO CORE FROM
C      DISK THE SECOND OVERLAY ON TOP OF THE FIRST OVERLAY.
C      IN THE SECOND OVERLAY THE SINE OF THE ANGLE ALPHA IS
C      CALCULATED.
0011  CALL LINK('OVL2')
C      RETURNED TO RESIDENT AT LABEL 3 FROM OVERLAY 2 VIA A
C      CALL RETURN.
0012  3  CALL PRINT(BUFF3)
C      CALL THE RESIDENT SUBROUTINE "LINK" TO BRING INTO CORE FROM
C      DISK THE THIRD OVERLAY ON TOP OF THE SECOND OVERLAY. IN
C      THE THIRD OVERLAY THE SQUARE ROOT OF THE SINE OF THE ANGLE
C      ALPHA IS CALCULATED.
0013  CALL LINK('OVL3')
C      RETURNED TO RESIDENT AT LABEL 4 FROM OVERLAY 4 VIA A CALL RETURN
0014  4  CALL PRINT(BUFF4)
0015  GO TO 1
0016  END

```

ROUTINES CALLED:
PRINT, LINK

BLOCK	LENGTH
MAIN.	191 (000576)*
.SSSS.	4 (000010)
X	2 (000004)

COMPILER ---- CORE
 PHASE USED FREE
 DECLARATIVES 00368 16218
 EXECUTABLES 00540 16044
 ASSEMBLY 00955 18546

```

0001      SUBROUTINE PRINT (BUFF)
          C      THIS IS A RESIDENT SUBROUTINE TO BE CALLED FROM THE RESIDENT
          C      SECTION OR FROM THE OVERLAYS
          C      THIS SUBROUTINE IS USED TO PRINT OUT TEXT ON THE TELETYPE,
          C      MAINLY TO IDENTIFY THE SECTION WHICH IS BEING EXECUTED.
0002      INTEGER BUFF(10)
0003      WRITE(6,5) BUFF
0004      5      FORMAT(1H ,10A2)
0005      RETURN
0006      END

```

```

BLOCK      LENGTH
PRINT      46      (000134)*

```

```

**COMPILER **** CORE**
      PHASE      USED      FREE
DECLARATIVES 00446 16138
EXECUTABLES  00446 16138
ASSEMBLY      00901 18600

```

```

      .TITLE FORCE
      /
      / THE SOLE PURPOSE OF THIS ASSEMBLY LANGUAGE MODULE IS TO
      / FORCIBLY LOAD THE "F" FORMAT INPUT & OUTPUT CONVERSION
      / ROUTINES INTO THE RESIDENT AREA FOR USE BY OVERLAY 1 &
      / OVERLAY 4 RESPECTIVELY. THIS IS ACCOMPLISHED BY
      / DECLARING AS GLOBALS SDCI & SPCO. THESE TWO GLOBALS
      / WILL CAUSE THE LINKER TO LOAD FROM THE FORTRAN LIBRARY
      / THE TWO REQUIRED FORMAT CONVERSION ROUTINES. NOTE!
      / THESE TWO CONVERSIONS ARE NOT REQUIRED IN THE RESIDENT. BUT
      / IF THEY ARE NOT FORCIBLY LOADED IN THE RESIDENT THE
      / FORTRAN COMPILER WILL CAUSE THE LINKING OF TWO DUMMY CONVERSION
      / ROUTINES INSTEAD. THEN GLOBAL REQUESTS IN THE OVERLAY LINKS
      / WILL BE LINKED TO THE DUMMY ROUTINES AND AT EXECUTION TIME
      / WILL RESULT IN THE FATAL ERROR MESSAGE:
      /
      / PORT000000 LINKAGE ERROR (MISSING FORMAT CONVERSION ROUTINE)
      /
      .GLOBAL SPCO,SDCI
      .END

```

```

FORCE      MACRO VR04=14 03-AUG-72 14158 PAGE 1*
SYMBOL TABLE

```

```

SDCI = ***** G      SPCO = ***** G
. ABS. 000000      000
      000000      001

```

```

ERRORS DETECTED: 0
FREE CORE: 16589. WORDS
FORCE,LP1:FORCE

```

NOTE

Due to the length of the comment lines in the FORCE module, only the source listing is shown above, although the block summary information is the result of compiling FORCE.

```

C      THIS IS OVERLAY 1.  IN THIS OVERLAY THE ANGLE ALPHA IS INPUT
C      FROM THE KEYBOARD WITH A FORMAT OF F5.2.
0001      COMMON /X/A
0002      INTEGER BUFF5(10), BUFF6(10)
0003      DATA BUFF5/'OV','ER','LA','Y',6*1 1/
0004      DATA BUFF6/'IN','PU','T',5015,6*1 1/
C      CALL THE RESIDENT SUBROUTINE "PRINT" TO PRINT "OVERLAY:"
C      TO INDICATE THAT CURRENTLY OVERLAY 1 IS BEING EXECUTED.
0005      CALL PRINT(BUFF5)
C      CALL THE RESIDENT SUBROUTINE "PRINT" TO PRINT "INPUT"
C      TO INDICATE TO THE OPERATOR THAT THE COMPUTER IS WAITING
C      FOR THE VALUE OF THE ANGLE ALPHA FROM THE KEYBOARD.
0006      CALL PRINT(BUFF6)
C      READ INTO THE VARIABLE "A" IN THE LABELLED COMMON AREA THE
C      ANGLE ALPHA FROM THE KEYBOARD.
0007      READ(6,6) A
0008      FORMAT(F5.2)
C      NOTE! THE CONTENTS OF BUFF6 IS BEING DYNAMICALLY CHANGED
C      AT RUN TIME. THIS WILL SHOW THAT OVERLAYS ARE COMPLETELY
C      REFRESHED ON THE NEXT CALL. THIS IMPLIES THAT NO DATA
C      COMMUNICATION CAN BE ACHIEVED BETWEEN OVERLAYS OTHER
C      THEN THROUGH RESIDENT COMMON.
0009      BUFF6(1)='O'
0010      BUFF6(2)='K'
0011      BUFF6(3)=' '
C      CALL THE RESIDENT SUBROUTINE "PRINT" TO PRINT "O K"
C      TO INDICATE TO THE OPERATOR THAT THE DATA HAVE BEEN
C      RECEIVED.
0012      CALL PRINT(BUFF6)
C      CALL "RETURN" TO REENTER THE RESIDENT ROUTINE WHERE THE LAST
C      CALL FOR "LINK" WAS ISSUED FROM THE RESIDENT. IT IS
C      AT LABEL 2.
0013      CALL RETURN
0014      END

```

ROUTINES CALLED:
PRINT , RETURN

BLOCK	LENGTH
MAIN.	170 (000524)*
X	2 (000004)

COMPILER ----- CURE

PHASE	USED	FREE
DECLARATIVES	00366	16218
EXECUTABLES	00536	16048
ASSEMBLY	00986	16515

```

C      THIS IS OVERLAY 2.  IN THIS OVERLAY THE SINE OF THE ANGLE
C      ALPHA IS CALCULATED.
0001      COMMON /X/A
0002      COMMON B
0003      INTEGER BUFF7(10)
0004      DATA BUFF7/'OV',IERI,ILA',IY2I,6*1 1/
C      CALL THE RESIDENT SUBROUTINE "PRINT" TO PRINT
C      "OVERLAY2" TO INDICATE THAT CURRENTLY OVERLAY2 IS BEING EXECUTED
0005      CALL PRINT(BUFF7)
C      CALL THE FORTRAN LIBRARY ROUTINE "SIN", WHICH IS LINKED
C      TO THIS OVERLAY ONLY, TO CALCULATE THE SINE OF THE ANGLE
C      ALPHA.  THE ANGLE ALPHA IS IN VARIABLE "A" AND THE SINE OF
C      ALPHA WILL BE PUT IN VARIABLE "B".
C      "A" IS IN THE LABELLED COMMON AREA AND "B" IS IN THE
C      BLANK COMMON AREA.
0006      B=SIN(A)
C      CALL "RETURN" TO REENTER THE RESIDENT ROUTINE WHERE THE LAST
C      CALL FOR "LINK" WAS ISSUED FROM THE RESIDENT IT IS AT LABEL3.
0007      CALL RETURN
0008      END

```

ROUTINES CALLED:
PRINT , SIN , RETURN

BLOCK		LENGTH
MAIN.	06	(000204)*
.3333.	2	(000004)
X	2	(000004)

COMPILER *** CORE**

PHASE	USED	FREE
DECLARATIVES	00366	16218
EXECUTABLES	00458	16126
ASSEMBLY	00891	18610

```

C      THIS IS OVERLAY 3.  IN THIS OVERLAY THE SQUARE ROOT OF THE SINE
C      OF THE ANGLE ALPHA IS CALCULATED.
0001      COMMON B,C
0002      INTEGER BUFF8(10)
0003      DATA BUFF8/'OV1','ER1','LA1','Y31','0*1' /
C      CALL THE RESIDENT SUBROUTINE "PRINT" TO PRINT "OVERLAY3"
C      TO INDICATE THAT CURRENTLY OVERLAY 3 IS BEING EXECUTED.
0004      CALL PRINT(BUFF8)
C      CALL THE FORTRAN LIBRARY ROUTINE "SQRT", WHICH IS LINKED TO
C      THIS OVERLAY ONLY, TO CALCULATE THE SQUARE ROOT OF THE SINE OF T
C      ANGLE ALPHA.  THE SINE OF ALPHA IS IN VARIABLE "B" AND THE
C      SQUARE ROOT OF IT WILL BE PUT IN VARIABLE "C".  BOTH ARE IN
C      THE BLANK COMMON AREA.
0005      C=SQRT(B)
C      CALL THE RESIDENT SUBROUTINE "LINK" TO BRING INTO CORE FROM
C      DISK THE FOURTH OVERLAY ON TOP OF THE THIRD OVERLAY, FROM
C      WHERE THE CALL WAS ISSUED.  IN THE FOURTH OVERLAY THE RESULTS
C      ARE GOING TO BE PRINTED ON THE TELETYPE.
0006      CALL LINK('OVL4')
0007      END

```

ROUTINES CALLED:
PRINT , SQRT , LINK

BLOCK	LENGTH
MAIN. 70	(000214)*
.SSSS. 4	(000010)

COMPILER ** CORE**

PHASE	USED	FREE
DECLARATIVES	00366	16218
EXECUTABLES	00450	16134
ASSEMBLY	00890	18611

```

      C      THIS IS OVERLAY 4. IN THIS OVERLAY THE RESULTS ARE PRINTED ON
      C      THE TELETYPE.
0001      COMMON /X/A
0002      COMMON B,C
0003      INTEGER BUFF9(10)
0004      DATA BUFF9/'OV1','ER1','LA1','Y41',6*1 /
      C      CALL THE RESIDENT SUBROUTINE "PRINT" TO PRINT "OVERLAY4" TO
      C      INDICATE THAT CURRENTLY OVERLAY 4 IS BEING EXECUTED.
0005      CALL PRINT(BUFF9)
      C      PRINT THE RESULTS ON THE TELETYPE. ALL WITH THE FORMAT
      C      OF F5.2. THE ANGLE ALPHA IS IN VARIABLE "A", THE SINE OF
      C      THE ANGLE ALPHA IS IN VARIABLE "B", THE SQUARE ROOT OF
      C      THE ANGLE ALPHA IS IN VARIABLE "C".
      C      VARIABLE "A" IS IN THE LABELLED COMMON AREA AND VARIABLES
      C      "B" AND "C" ARE IN THE BLANK COMMON AREA.
0006      WRITE(6,7) A,B,C
0007      7      FORMAT(3H0A=,F5.2,6X,7H8IN(A)=,F5.2,6X,12H8SQRT SIN(A)=,F5.2/)
      C      CALL "RETURN" TO REENTER THE RESIDENT ROUTINE WHERE THE LAST
      C      CALL FOR "LINK" WAS ISSUED FROM THE RESIDENT. IT IS AT LABEL4.
0008      CALL RETURN
0009      END

```

ROUTINES CALLED:
PRINT , RETURN

BLOCK	LENGTH
MAIN. 100	(000324)*
.S333. 4	(000010)
X 2	(000004)

COMPILER ----- CORE

PHASE	USED	FREE
DECLARATIVES	00366	16218
EXECUTABLES	00459	16125
ASSEMBLY	01063	18438

The following output shows the disk directory before linking
the overlay system:

DIRECTORY DK0: [200,200]

03-AUG-72

FORCE .OBJ	1	03-AUG-72 <233>
RES .OBJ	3	03-AUG-72 <233>
PRINT .OBJ	2	03-AUG-72 <233>
OVL1 .OBJ	3	03-AUG-72 <233>
OVL2 .OBJ	2	03-AUG-72 <233>
OVL3 .OBJ	2	03-AUG-72 <233>
OVL4 .OBJ	2	03-AUG-72 <233>

TOTL BLKS: 15
TOTL FILES: 7

The process of linking this overlay system is shown below:
 The commands to the Linker create the resident load module and the four overlay load modules. Note the presence of the resident symbol table, the FORTRAN Library file, and the /OV switch while linking the overlay files.

```
SRUN LINK
LINK  V010A.2
#RES,LP:,SY:RES<RES/OV:4,PRINT,FORCE,FTNLIB/L/E
LINK  V010A.2
#OVL1,LP:<OVL1,FTNLIB/L/E
LINK  V010A.2
#OVL2,LP:<OVL2,FTNLIB/L/E
LINK  V010A.2
#OVL3,LP:<OVL3,FTNLIB/L/E
LINK  V010A.2
#OVL4,LP:<OVL4,FTNLIB/L/E
LINK  V010A.2
#
```

The following disk directory shows the results of linking the overlay system:

DIRECTORY DK0: [233,233]

03-AUG-72

RES	.STB	4	03-AUG-72	<233>
RES	.LDA	20	03-AUG-72	<233>
OVL1	.LDA	2	03-AUG-72	<233>
OVL2	.LDA	4	03-AUG-72	<233>
OVL3	.LDA	3	03-AUG-72	<233>
OVL4	.LDA	1	03-AUG-72	<233>
FORCE	.OBJ	1	03-AUG-72	<233>
RES	.OBJ	3	03-AUG-72	<233>
PRINT	.OBJ	2	03-AUG-72	<233>
OVL1	.OBJ	3	03-AUG-72	<233>
OVL2	.OBJ	2	03-AUG-72	<233>
OVL3	.OBJ	2	03-AUG-72	<233>
OVL4	.OBJ	2	03-AUG-72	<233>

TOTL BLKS: 49
 TOTL FILES: 13

Shown below are the load maps for the overlay system.

Load Map	<u>Notes</u>
LOAD MAP RES .LDA 15102127 03-AUG-72	
TRANSFER ADDRESS: 141210	
LOW LIMIT: 141210	
HIGH LIMIT: 157400	

MODULE MAIN.	Resident main program.
SECTION ADDRESS SIZE	
< ABS.> 000000 000000	
< > 141210 000576	
MAIN. 141210	
< SSSS.> 142000 000010	Blank common
< X > 142016 000004	Labeled common

MODULE PRINT	
SECTION ADDRESS SIZE	Resident Subroutine PRINT
< > 142022 000134	
PRINT 142022	

MODULE FORCE	
SECTION ADDRESS SIZE	Resident dummy
< > 142156 000000	Subroutine FORCE

.	
.	
.	Resident OTS routine

MODULE \$DCO04	
SECTION ADDRESS SIZE	
< > 142226 001614	
\$DCO 142240	\$ECO 142270
\$GCO 142226	\$FCO 142234

MODULE \$DCI01	
SECTION ADDRESS SIZE	
< > 144042 001400	
\$DCI 144050	\$HCI 144042

.	
.	\$DCI01 and \$DCO04
.	two forcibly loaded
.	conversion packages

MODULE \$DIOUD	
SECTION ADDRESS SIZE	Dummy unformatted
< > 151724 000014	I/O routines
\$DIOUD 151724	

MODULE \$DIOUD	
SECTION ADDRESS SIZE	Dummy direct-access
< > 151740 000014	I/O routine
\$DIOUD 151740	

Load MapNotes

```

:
:
*****
MODULE  $LNK03
SECTION  ADDRESS  SIZE
<      > 153402  000442
          LINK    153430      RETURN  153634      RUN      153402
*****
MODULE  $POL07
SECTION  ADDRESS  SIZE
<      > 154044  000004
          $EAE    154050      $POLSH  154044      $V20A   154044
*****
MODULE  $SEQ02
SECTION  ADDRESS  SIZE
<      > 154050  000006
          $SEQ    154050
*****
MODULE  $OLCO
SECTION  ADDRESS  SIZE
<      > 154056  000014
          $LCO    154056
*****
MODULE  $OICO
SECTION  ADDRESS  SIZE
<      > 154072  000014
          $ICO    154072      $OCO    154072
*****
MODULE  $OICI
SECTION  ADDRESS  SIZE
<      > 154106  000014
          $ICI    154106      $UCI    154106
*****
MODULE  $OLCI
SECTION  ADDRESS  SIZE
<      > 154122  000014
          $LCI    154122
*****
:
:
*****
MODULE  $OTV04
SECTION  ADDRESS  SIZE
<      > 155634  000032
          $ERRNK  155650      $EXSW   155646      $NAMC   155642
          $OTSV   155634      $SEQC   155644
***
:
:
*****
MODULE  $DEV09
SECTION  ADDRESS  SIZE
<      > 156462  000530
          $DEVTB  156460
*****
MODULE  $IBF02
SECTION  ADDRESS  SIZE
<      > 157212  000246
          $IOBUF  157214
*****

```

LINK subroutine

entry points: LINK,
RETURN and RUNFour dummy
format
conversion
routinesFORTRAN impure
area is \$OTSVFORTRAN device
table is \$DEVTBFORTRAN I/O
buffer

Load MapNotes

LOAD MAP OVL1 .LDA 15103120 03-AUG-72

TRANSFER ADDRESS: 140312

LOW LIMIT: 140312

HIGH LIMIT: 141206

MODULE \$\$\$N.

Resident main program

SECTION	ADDRESS	SIZE
< ABS.>	000000	000000
< >	141210	000576
\$\$\$N.	141210	
< \$\$\$S.>	142000	000010
<X >	142010	000004

MODULE PRINT

Resident subroutine
PRINT

SECTION	ADDRESS	SIZE
< >	142022	000134
PRINT	142022	

.
.
.Resident subroutines
exactly the same load map
as for the resident program
(RES.LDA)

MODULE MAIN.

Overlay 1 main program

SECTION	ADDRESS	SIZE
< >	140312	000524
MAIN.	140312	

MODULE SPUT03

OTS routines included in
overlay 1 but not in resident
overlay
\$PUT03 and \$\$\$S08

SECTION	ADDRESS	SIZE
< >	141036	000012
SPUT1	141044	
SPUT4	141036	

SPUT2	141044
SPUT5	141036

MODULE \$\$\$S08

SECTION	ADDRESS	SIZE
< >	141050	000136
\$\$\$S1	141050	

\$\$\$S2	141056
\$\$\$S3	141064

Load MapNotes

LOAD MAP OVL2 .LDA 15104:22 03-AUG-72

TRANSFER ADDRESS: 136500

LOW LIMIT: 136500

HIGH LIMIT: 141200

MODULE SSN.

SECTION	ADDRESS	SIZE
< ABS.>	000000	000000
< >	141210	000570
SSN.	141210	
< SSS.>	142000	000010
< X >	142010	000004

MODULE PRINT

SECTION	ADDRESS	SIZE
< >	142022	000134
PRINT	142022	

.

.

.

MODULE MAIN.

SECTION	ADDRESS	SIZE
< >	136500	000204
MAIN.	136500	

MODULE SPOP03

SECTION	ADDRESS	SIZE
< >	130704	000020
SPOP3	130720	
SPOP4B	130700	

MODULE SSIN04

SECTION	ADDRESS	SIZE
< >	130724	000354
COS	130724	

.

.

.

Resident main program

Resident subroutine
PRINTResident subroutine
exactly the same load map
as for the resident
program (RES.LDA)

Overlay 2 main program

Overlay 2 OTS routines follow:

SPOP4	136704	SPUP4A	136712
SPOP5	136704		

SIN	136700
-----	--------

Overlay 2 subroutines

Load MapNotes

LOAD MAP UVL3 .LDA 15105118 03-AUG-72

TRANSFER ADDRESS: 137436

LOW LIMIT: 137436

HIGH LIMIT: 141206

MODULE SSSN.

SECTION

< . ABS. > ADDRESS SIZE

< > 000000 000000

< > 141210 000576

< . SSSN. > 141210

< . SSSN. > 142000 000010

< X > 142010 000004

MODULE PRINT

SECTION

< > ADDRESS SIZE

< > PRINT 142022 000134

.

.

.

MODULE MAIN.

SECTION

< > ADDRESS SIZE

< > MAIN. 137436 000214

< > 137436

MODULE SPOP03

SECTION

< > ADDRESS SIZE

< > SPOP3 137652 000020

< > SPOP4B 137654

MODULE SSWT03

SECTION

< > ADDRESS SIZE

< > SSWT 137672 000136

..

.

.

Resident main program

Resident subroutine
PRINTResident subroutines
exactly the same as
for the resident program
(RES.LDA)

Overlay 3 main program

Overlay 3 OTS routines follow:

SPOP4 137652 SPOP4A 137660
SPOP5 137652

Overlay 3 subroutines

Load Map

LOAD MAP OVL4 .LDA 15106112 03-AUG-72

TRANSFER ADDRESS: 140662

LOW LIMIT: 140662

HIGH LIMIT: 141206

MODULE SSSN.

SECTION	ADDRESS	SIZE
<, ABS.>	000000	000000
<	141210	000576
<, SSSN.>	141210	000010
<X>	142010	000004

Resident main program

MODULE PRINT

SECTION	ADDRESS	SIZE
<	142022	000134
PRINT	142022	

Resident subroutine
PRINT

.
.
.

Resident subroutines
exactly the same as for
the resident program
(RES.LDA)

MODULE MAIN.

SECTION	ADDRESS	SIZE
<	140662	000324
MAIN.	140662	

Overlay 4 main program

Shown below is the run-time output of this overlay system. The output clearly traces the control path through the system.

SRUN RES

RES IDENT
OVERLAY1
INPUT?
1.00

O K

LABEL2
OVERLAY2
LABEL3
OVERLAY3
OVERLAY4

A= 1.00 SIN(A)= 0.84 SQR SIN(A)= 0.92

LABEL4
RES IDENT
OVERLAY1
INPUT?
1.57

O K

LABEL2
OVERLAY2
LABEL3
OVERLAY3
OVERLAY4

A= 1.57 SIN(A)= 1.00 SQR SIN(A)= 1.00

LABEL4
RES IDENT
OVERLAY1
INPUT?

IC

.KI

\$FINISH

TIME:-15:09:22

7.12 CALL RUN SUBROUTINE

The FORTRAN LINK subroutine can be called to load and execute another, unrelated, program. The form of the subroutine call is as follows:

```
CALL RUN ('dev:file.ext[uic]')
```

A complete filename specification is passed as an alphanumeric string to the subroutine. (See the description of character string argument storage formats in Section 6.5.10.) Programs loaded using CALL RUN are not in an overlay relationship to the calling program; the program need not be a FORTRAN program.

The CALL RUN performs the following functions:

- a. All files opened by the FORTRAN OTS are closed, and
- b. the DOS Monitor RUN EMT is invoked to load and transfer control to the requested program.

Loading a program by means of CALL RUN does not modify memory locations whose addresses are greater than the upper limit of the program being loaded. Thus, it is possible for two FORTRAN programs to pass data in common blocks in the high end of memory.

Common blocks used for communication between independently linked programs must be assigned the same addresses in both programs. A recommended technique is to use a BLOCK DATA subprogram to define the common blocks passed between programs. The BLOCK DATA subprogram is then linked first, forcing the common blocks to the high end of memory. The symbol table file resulting from this linking (BLOCK.STB in the following example), is then used in linking the calling and called programs. In the following example, the programs CALLER and CALLED use this method to force the blank common and C1 common blocks into the same addresses at the high end of memory.

BLOCK DATA Subprogram

FORTRAN V304A

15:48:51

12-JUL-72

PAGE

1

```
C      BLOCK.FTN
C      THIS PROGRAM DECLARES ALL THE COMMON
C      WHICH IS TO BE SHARED BY CALLER.FTN & CALLED.FTN
C
0001      BLOCK DATA
0002      COMMON /C1/ IC1,JC1,KC1
0003      COMMON IA(20),IBLANK
0004      DATA IC1,JC1,KC1,IBLANK/ -3,-4,-5,5/
0005      END
```

BLOCK		LENGTH
DATA.	3	(000000)*
,\$\$\$\$.	42	(000124)
C1	6	(000014)

```
**COMPILER ----- CORE**
      PHASE      USED  FREE
DECLARATIVES 00366 17016
EXECUTABLES  00461 16921
ASSEMBLY     00821 19478
```

CALLER Subprogram

FORTRAN V004A

15:49:01

12-JUL-72 PAGE 1

```

C      CALLER.FTN
C      THIS PROGRAM SETS UP SOME VALUES IN COMMON,
C      WRITES A FILE(AND LEAVES IT OPEN),
C      AND THEN INVOKES CALLED.FTN USING 'CALL RUN'.
C
C      THESE COMMON BLOCKS ARE SHARED BY CALLED.FTN & CALLER.FTN
0001      COMMON IA(20),IBLANK
0002      COMMON /C1/ IC1,JC1,KC1
C
C      THIS IS "LOCAL" COMMON (TO CALLER.FTN)
0003      COMMON /COM111/ MINE,MINE1(300)
C
0004      WRITE(5,530) IC1,JC1,KC1,IBLANK
0005      530  FORMAT(' CALLER.FTN'/' VALUES FOUND IN COMMON:',4I8)
C
C      PUT SOME VALUES IN COMMON AND FILE
0006      IC1=20
0007      IBLANK=50
0008      DO 20 I=1,20
0009      20  IA(I)=2*I
0010      CALL SETFIL(1,'CALLER.FIL')
0011      DEFINE FILE 1(5,20,U,IVAR)
0012      WRITE(1'1)IA
C
0013      WRITE(5,510)
0014      510  FORMAT('0VALUES IN THE FILE:')
0015      WRITE(5,520)IA
0016      520  FORMAT(10I6)
0017      DO 40 I=1,20,5
0018      40  IA(I)=IA(I)/7+1
0019      WRITE(5,530)IC1,JC1,KC1,IBLANK
0020      530  FORMAT('0VALUES PASSED IN COMMON:',4I8/)
0021      WRITE(5,520)IA
C
C
0022      CALL RUN('CALLED')
0023      END

```

ROUTINES CALLED:
SETFIL, RUN

BLOCK	LENGTH
MAIN.	302 (001134)*
.9SSS.	42 (000124)
C1	6 (000014)
COM111	602 (002264)

COMPILER ----- CORE
 PHASE USED FREE
 DECLARATIVES 00471 16911
 EXECUTABLES 00607 16775
 ASSEMBLY 01226 19073

CALLED Subprogram

FORTRAN V004A

15:49:23

12-JUL-72

PAGE

1

```

      C      CALLED,FTN
      C      THIS IS THE PROGRAM CALLED BY CALLER (GOT THAT?)
      C
      C      THESE COMMON BLOCKS ARE SHARED BY CALLED,FTN & CALLER,FTN
0001      COMMON IA(20),IBLANK
0002      COMMON /C1/ IC1,JC1,KC1
      C
      C      THIS IS "LOCAL" COMMON (TO CALLED,FTN)
0003      COMMON /COM999/ MINE3,MYOWN(500)
      C
0004      INTEGER IB(20)
0005      WRITE(5,500) IC1,JC1,KC1,IBLANK
0006      500  FORMAT(' CALLED,FTN'/' VALUES FOUND IN COMMON:',4I6/)
0007      WRITE(5,520) IA
      C
0008      CALL SETFIL(2,'CALLER.FIL')
0009      DEFINE FILE 2(5,20,U,JJVAR)
0010      READ(2:1)IB
      C
0011      WRITE(5,510)
0012      510  FORMAT('VALUES IN THE FILE:')
0013      WRITE(5,520)IB
0014      520  FORMAT(12I6)
      C
      C
0015      STOP
0016      END
```

ROUTINES CALLED:

SETFIL

BLOCK	LENGTH
MAIN.	217 (000662)*
.SSSS.	42 (000124)
C1	6 (000014)
COM999	1002 (003724)

```

**COMPILER ----- CORE**
      PHASE      USED  FREE
DECLARATIVES 00471 16911
EXECUTABLES  00527 16855
ASSEMBLY     01138 19161
```

BATCH Command Strings to Compile, Link and
Execute the Example

\$BA DT:LNKEX2.BAT,KB:

\$JOB LNKEX2(200,200)

DATE:-12-JUL-72

TIME:-15:48:40

\$RUN FORTRN

FORTRAN V004A

#BLOCK,LP:<BI:BLOCK

#CALLER,LP:<BI:CALLER

#CALLED,LP:<BI:CALLED

#

\$RUN LINK

LINK V010A.2

#BLOCK,LP:,SY:BLOCK<BLOCK/E

LINK V010A.2

#CALLER,LP:<BLOCK.STB,CALLER,BLOCK,FTNLIB/L/E

LINK V010A.2

#CALLED,LP:<BLOCK.STB,CALLED,FTNLIB/L/E

LINK V010A.2

#

\$RUN CALLER

I350 000000

\$FINISH

TIME:-15:52:23

MONITOR V006A

\$

Load MapsNotes

LOAD MAP BLOCK.LDA 15:49:46 12-JUL-72

TRANSFER ADDRESS: 000001
LOW LIMIT: 157320
HIGH LIMIT: 157460Load map
of BLOCK DATA
subprogram*****
MODULE DATA.
SECTION ADDRESS SIZE
<.ABS.> 000000 000000
< > 157320 000000
<.SSSS.> 157320 000124
<C1 > 157444 000014

LOAD MAP CALLER.LDA 15:50:10 12-JUL-72

TRANSFER ADDRESS: 137032
LOW LIMIT: 137032
HIGH LIMIT: 157316*****
MODULE SSSA.
SECTION ADDRESS SIZE
<.ABS.> 000000 000000
< > 157320 000000
<.SSSS.> 157320 000124
<C1 > 157444 000014shared common areas
defined by BLOCK DATA
subprogramblank common
labeled common block C1*****
MODULE MAIN.
SECTION ADDRESS SIZE
< > 137032 001134
MAIN. 137032
<COM111> 140166 002264

CALLER MAIN program

labeled common block COM111

MODULE DATA.
SECTION ADDRESS SIZE
< > 142452 000000

OTS routines

MODULE SLNK03
SECTION ADDRESS SIZE
< > 153224 000442
LINK 153252

RETURN 153456

RUN 153224

RUN Subroutine
entry points: LINK, RUN
and RETURN.

Load Maps

Notes

LOAD MAP CALLED.LDA 15:51:20 12-JUL-72

TRANSFER ADDRESS: 136370
LOW LIMIT: 136370
HIGH LIMIT: 157316

MODULE DATA,

SECTION	ADDRESS	SIZE
<.ABS.>	000000	000000
< >	157320	000000
<.SSSS.>	157320	000124
<C1 >	157444	000014

MODULE MAIN,

SECTION	ADDRESS	SIZE
< >	136370	000662
MAIN,	136370	
<COM999>	137252	003724

.
.
.

Shared common areas
defined by BLOCK DATA
subprogram

CALLED Main Program

labeled common block
COM999

OTS Routines

Program Output

CALLER.FTN

VALUES FOUND IN COMMON: -3 -4 -5 5

VALUES IN THE FILE:

2	4	6	8	10	12	14	16	18	20
22	24	26	28	30	32	34	36	38	40

VALUES PASSED IN COMMON: 20 -4 -5 50

1	4	6	8	10	2	14	16	18	20
4	24	26	28	30	5	34	36	38	40

CALLED.FTN

VALUES FOUND IN COMMON: 20 -4 -5 50

1	4	6	8	10	2	14	16	18	20
4	24	26	28	30	5	34	36	38	40

VALUES IN THE FILE:

2	4	6	8	10	12	14	16	18	20
22	24	26	28	30	32	34	36	38	40

CHAPTER 8

FORMAT STATEMENTS

Data transmission statements govern the transfer of data between internal storage and either peripheral devices or other locations in the computer memory. These statements are divided into four categories:

- | | |
|--|--|
| a. Data description statements: | FORMAT (described in this Chapter) and DEFINE FILE (Section 9.4.1) |
| b. Input/Output statements: | READ, WRITE and PRINT (Chapter 9) |
| c. Device control statements for magnetic tape and disk usage: | FIND, BACKSPACE, REWIND and END FILE (Sections 9.4.2, 9.6.1, 9.6.2, and 9.6.3) |
| d. Statements specifying data transmission between internal storage areas: | ENCODE or DECODE (Section 9.7) |

Chapter 8 describes the construction of FORMAT Statements. Chapter 9 describes the remaining data transmission statements.

8.1 FORMAT STATEMENT

The FORMAT statement describes the format in which one or more records are transmitted. FORMAT statements are non-executable statements used in conjunction with an Input/Output statement or with an ENCODE or DECODE statement.

FORMAT statements are of the form:

statement number `FORMAT(S1,S2,...Sn/S11,S21,...Sn1/...)`

where a statement number must be specified. (FORMAT statements are referenced by statement number.) Each S is a data field specification and the slash (/) character indicates the start of a new record. The closing parenthesis at the end of the FORMAT statement also terminates a record.

A record can be a punched card (80 columns), a line of Teletype (truncated to 72 characters) or line printer output (133 characters maximum), or a tape or disk string terminated by a carriage return/line feed code. Line feed is the accepted terminator (see Section 10.2).

Each record description may consist of one or more field specifications; a field being a consecutive series of characters within the record. Field specifications are separated by commas within the FORMAT statement (consecutive commas are not allowed).

FORMAT statements must be placed in the executable part of the source program. Unless the FORMAT statement contains only alphanumeric data for direct transmission, it is used in conjunction with the list of a data transmission statement. The FORMAT statement indicates the form in which the variable(s) specified in the data transmission statement are to be read or written.

During data transmission, the object program scans the indicated FORMAT statement. If one or more specification(s) for a numeric field, for example, are present and the data transmission statement contains numeric data items to be transmitted, transmission takes place according to the format specification. Execution of the data transmission statement is terminated when all items in the data transmission list have been handled. The FORMAT statement can contain specifications for more items than are contained in a given data transmission statement, in which case excess format specifications are ignored. Conversely, the FORMAT statement can contain specifications for fewer items than are contained in the data transmission list, in which case the set of format specifications beginning with the last left parenthesis or last slash is repeated as often as necessary to supply the needed number of data formats.

A field specification within a FORMAT statement is of the form `Xw.d` where `X` is a format code, `w` indicates the field size, and `.d` (where present) indicates the number of characters within the field that occur following the decimal point. The following types of format conversion codes may appear in a FORMAT statement:

- | | |
|--------------------------------|--------------------------------|
| a. numeric: | Iw, Ow, Fw.d, Ew.d, Dw.d, Gw.d |
| b. numeric with scale factors: | nP |
| c. logical: | Lw |
| d. alphanumeric and editing: | Aw, nH, '...', nX, Tw |

8.2 I FORMAT CONVERSIONS

A specification of the form:

Iw

is used to transmit decimal integer values. The corresponding element in the data transmission list must be an integer or byte variable.

On output, a field w characters long is reserved for the value to be output. The value is transmitted into this w-position field right-justified. Where a minus sign is expected in the output, one position must be included in w for that sign. Since a plus sign is suppressed, no allowance need be made for it. Where the field specified for the output value is too small, the entire output field is filled with asterisks. For example:

<u>Format</u>	<u>Internal Value</u>	<u>External Representation</u>
I3	284	284
I4	-284	-284
I5	174	~~00~~ 174
I2	3244	**
I3	-473	***
I7	29.443	not permitted, error

On input, a field of w positions is read as it appears. The external data must be a decimal integer and not contain a decimal point or exponent field. Blanks are interpreted as zeroes. Leading blanks are ignored. An all blank field is interpreted as a 0 on input. An integer overflow occurs if the value being read is beyond the range -32768 to 32767. For example:

<u>Format</u>	<u>External Value</u>	<u>Internal Representation</u>
I4	2788	2788
I3	-26	-26
I9	~~000000~~ 312	312
I9	312 ~~000000~~	not permitted, error; system attempts to input 312000000

8.3 O FORMAT CONVERSIONS

A specification of the form:

Ow

is used to transmit octal integer values. The corresponding element in the data transmission list may be an integer or byte variable.

On output a field w characters long is reserved for the value. The value is transmitted into this field right-justified. Where the field specified is too small, the entire output field is filled with asterisks. For example:

<u>Format</u>	<u>Internal (Decimal) Value</u>	<u>External Representation</u>
O6	32767	Ø77777
O6	-32768	1ØØØØØ
O2	14261	**
O4	27	ØØ33
O5	13.52	not permitted, error

On input, a field of w positions is read as it appears. The external values must not contain a sign, a decimal point, exponent field or the digits 8 or 9. Blanks are interpreted as zeroes, with an all blank field interpreted as a Ø on input. Leading blanks are ignored. The value is read into the first word of the data transmission element specified in the input statement. For example:

<u>Format</u>	<u>External Value</u>	<u>Internal Octal Representation</u>
O5	32767	32767
O4	16234	1623
O6	13ØØØØ	13ØØØØ
O3	97	not permitted, error

8.4 F FORMAT CONVERSIONS

A specification of the form:

Fw.d

is used to transmit real or complex values. The corresponding element in the data transmission list must be a real or complex variable.

On output a field w characters long is reserved for the value, which is transmitted into this field right-justified. The total field length reserved (w) includes a decimal point, at least one digit to the left of the decimal point, a sign (only minus signs are printed on output), and d decimal digits following the decimal point. That is to say, $w \geq 3$. Where the field length specified for the output value is too small, the entire output field is filled with asterisks. For F conversion output, the data value after scaling must be within the range 0.14×10^{-38} to 1.7×10^{38} . For example:

<u>Format</u>	<u>Internal Value</u>	<u>External Representation</u>
F8.5	2.3547188	02.35472
F9.3	8789.7361	08789.736
F2.3	51.44	**
F10.4	-23.24352	00-23.2435
F5.2	325.013	*****
F5.2	-.2	-0.20

On input, a field of w positions is read as it appears. The external data must be a decimal number with or without a decimal point or exponent field. Blanks are interpreted as zeroes. Leading blanks are ignored. An all blank field is interpreted as 0. Where no decimal point is encountered within the first w positions, the w digits are accepted as the input value and a decimal point is inserted d positions from the right. A decimal point found in the input value overrides the format specification. For example:.

<u>Format</u>	<u>External Value</u>	<u>Internal Representation</u>
F8.5	123456789012	123.45678
F8.5	12345.6700	12345.67
F8.5	24.77E+2	2477.0
F5.2	1234567.88	123.45

8.5 E FORMAT CONVERSIONS

A specification of the form:

Ew.d

is used to transmit real or complex values, with a decimal point and E exponent field. The corresponding element in the data transmission list must be a real or complex variable.

On output a field w characters long is reserved for the value, which is transmitted into this field right-justified. The total field length reserved (w) includes a decimal point, at least one digit to the left of the decimal point, a sign (only minus signs are printed on output), d decimal digits following the decimal point, and four positions for the exponent representation. That is to say, $w \geq d+7$ in most cases. Where the field length specified for the output value is too small, the entire output field is filled with asterisks. For E conversion output, the data value, after scaling, must be within the range 0.14×10^{-38} to 1.7×10^{38} . For example:

<u>Format</u>	<u>Internal Value</u>	<u>External Representation</u>
E9.2	475867.222	00.48E06
E12.3	475867.222	0000.476E06
E5.3	56.12	*****

On input, a field of w positions is read as it appears. The external data must be a decimal number with or without a decimal point or exponent field. Blanks are interpreted as zeroes. Leading blanks are ignored. An all blank field is interpreted as 0. Where no decimal point is encountered within the first w positions, the w digits are accepted as the input value and a decimal point is inserted d positions from the right, and a zero exponent is assumed. A decimal point found in the input value overrides the format specification. For example:

<u>Format</u>	<u>External Value</u>	<u>Internal Representation</u>
E9.3	734.432E8	734.432E8
E12.4	001022.43E-6	1022.43E-6
E15.3	52.3759663	52.3759663
E10.4	01971.3213E12	1971.3213
E8.1	123456789012	1234567.8E0
E12.5	210.5271D+10	210.5271E10

8.6 D FORMAT CONVERSIONS

A specification of the form:

Dw.d

is used to transmit real values, with a decimal point and D exponent field. The corresponding element in the data transmission must be a real or double-precision variable.

On output a field w characters long is reserved for the value, which is transmitted into this field right-justified. The total field length reserved (w) includes a decimal point, at least one digit to the left of the decimal point, a sign (only minus signs are printed on output), d decimal digits following the decimal point, and four positions for the exponent representation. That is to say, $w \geq d+7$ in most cases. Where the field length specified for the output value is too small, the entire output field is filled with asterisks. For E conversion output, the data value, after scaling, must be within the range 0.14×10^{-38} to 1.7×10^{38} . For example:

<u>Format</u>	<u>Internal Value</u>	<u>External Representation</u>
D14.3	0.363	000000.363D000
D23.12	5413.87625793	000000.541387625793D004
D9.6	1.3	*****

On input, a field of w positions is read as it appears. The external data must be a decimal number with or without a decimal point or exponent field. Blanks are interpreted as zeroes. Leading blanks are ignored. An all blank field is interpreted as 0. Where no decimal point is encountered within the first w positions, the w digits are accepted as the input value and a decimal point is inserted d positions from the right, and a zero exponent is assumed. A decimal point found in the input value overrides the format specification. For example:

<u>Format</u>	<u>External Value</u>	<u>Internal Representation</u>
D10.2	1234500000	12345000.0D0
D10.2	00123.45000	123.45 D0
D15.3	367.4981763D+04	3.674981763D+06

8.7 G FORMAT CONVERSIONS

A specification of the form:

Gw.d

is used to transmit real or complex data.

On output a field w characters long is reserved for the value, which is transmitted into this field right-justified. The form of the output conversion depends upon the individual variable in the data transmission list. The output conversion format is a function of the magnitude of the data being converted, as described in Table 8-1.

Table 8-1

Floating-Point Magnitudes and Resulting G Format Conversions

<u>Data Magnitude</u>	<u>Resulting Conversion</u>
$0.1 \leq m < 1$	F(w-4).d, 4X
$1 \leq m < 10$	F(w-4).(d-1), 4X
.	.
$10^{d-2} \leq m < 10^{d-1}$	F(w-4).1, 4X
$10^{d-1} \leq m < 10^d$	F(w-4).0, 4X
All others	Ew.d

The total field length reserved (w) includes a decimal point, at least one digit to the left of the decimal point, a sign (only minus signs are printed on output), d decimal digits following the decimal point, and four positions for the exponent representation. That is to say, $w \geq d+7$ in most cases. Where the field length specified for the output value is too small, the entire output field is filled with asterisks. For example:

<u>Format</u>	<u>Internal Value</u>	<u>External Representation</u>
G12.5	-0.367451	-0.36745*****
G12.5	-3.67458	-3.6746*****
G12.5	36.7451	36.745*****
G12.5	357.458	367.46*****
G12.5	367.451	367.45*****

On input, a field of w positions is read as it appears. The external data must be a decimal number with or without a decimal point or exponent field. The rules for input conversion are the same as those for E format input conversion.

8.8 COMPLEX I/O

Complex data values are transmitted as two, independent, real quantities. The format specification consists of one repeated real specification or two successive real specifications of the form:

<u>Two Real Specifications</u>	<u>Repeated Real Specification</u>
(Fw.d,Fw.d)	2 (Fw.d)
(Ew.d,Ew.d)	2 (Ew.d)
(Gw.d,Gw.d)	2 (Gw.d)

Nothing is output between the two parts of a complex number unless specified by the format.

Output Conversion:

<u>Format</u>	<u>Internal Value</u>	<u>External Representation</u>
2 (F8.5)	2.3547188,3.456732	~~2.35472~~ 3.45673
(E9.2,E5.3)	475867.222,56.12	~~475.867E06~~ *****

Input Conversion:

<u>Format</u>	<u>External Value</u>	<u>Internal Representation</u>
(F8.5,F8.5)	1234567812345.67	123.45678,12345.67
(E9.1,E9.3)	734.432E8123456789012	734.432E8,1234567.8E-1

8.9 SCALE FACTORS

Scale factors of the form:

$$nP$$

can be specified for D, E, F, and G conversions. The number n is a signed or unsigned integer specifying the scale factor.

For E and D type output conversions, the basic real constant part of the output quantity is multiplied by 10^n and the exponent is reduced by n. For G type output conversion, the effect of the scale factor is suspended unless the magnitude of the value to be converted is outside the F conversion range.

For F, E, G and D type input conversions, where no exponent is found in the external value, and F type output conversions, the scale factor causes the external value to equal the internal value times 10^n .

For F, E, G and D type input conversions, the scale factor has no effect if there is an exponent in the external value.

Where no scale factor is specified, a scale factor of zero is assumed. Once a scale factor has been specified, it is carried onto all subsequent D, E, F, and G type conversions within the same format specification unless another scale factor is encountered. A zero scale factor can be reinstated by an explicit `0P` specification.

Output Conversions:

<u>Format</u>	<u>Internal Value</u>	<u>External Representation</u>
3PE15.3	-273.612	000-273.612E000
1PE15.3	-273.612	00000-2.736E002
1PE15.2	-273.612	000000-2.74E002
-1PE15.2	-273.612	000000-0.03E004

Input Conversions:

<u>Format</u>	<u>External Value</u>	<u>Internal Representation</u>
3PE10.5	00037.614	.037614
3PE10.5	0037.614E2	3761.4
-3PE10.5	0000037.614	37614.0

8.10 L FORMAT CONVERSIONS

A specification of the form:

Lw

is used to transmit the value of a logical variable.

On output a field w characters long is reserved for the value, which is transmitted into this field right-justified. The letter T or F is transmitted following w-1 blanks. For example:

<u>Format</u>	<u>Internal Value</u>	<u>External Representation</u>
L5	.TRUE.	0000T
L1	.FALSE.	F

On input, the first nonblank character in the data field must be a T or F; the value of the logical variable is stored as true or false, respectively. If the data field is blank or empty, a value of false is stored. Any other value in the data field causes an error diagnostic.

8.11 A FORMAT CONVERSIONS

Alphanumeric data can be transmitted in a manner similar to numeric data by use of a specification of the form:

Aw

A conversions are generally used where the alphanumeric data is generated or operated upon by the user program.

The field specified has length w. The alphanumeric characters are transmitted as the value of a variable in the data transmission list. The variable can be of any type. The value of w is limited to the maximum number of characters which can be stored in the space allotted for a single variable of the type specified. Table 8-2 indicates the number of alphanumeric characters which can be stored in a single variable of a given type.

Table 8-2

Alphanumeric Data Storage

<u>Variable Type</u>	<u>Number of Characters per Variable</u>
integer	2
byte	1
logical	2
real	4
double-precision	8
complex	8

For example, the sequence:

```
      READ(2,5) V
5     FORMAT (A4)
```

causes four characters to be read and stored in memory as the value of the real variable V.

If the value of w exceeds the amount specified in Table 8-2, the leftmost characters are lost on input, and on output the w characters appear right-justified in the external output field.

If the value of w is less than the number of characters which can be stored in the space allotted to a variable, on input the characters are left-justified and blank-filled on the right of each list item. On output the lower-order w characters in the variable are transmitted to the output field.

8.12 ALPHANUMERIC DATA WITHIN FORMAT SPECIFICATIONS

Alphanumeric data can be transmitted directly into or from a format specification by means of H (Hollerith) conversion or use of single quote delimiting characters. Direct transmission of alphanumeric data is performed where such data is not operated upon by the program; i.e., headings and explanations to be printed along with output data.

The H conversion is indicated by means of a specification of the form:

nH

where n indicates the number of characters to be transmitted. On output, the following n characters are sent to the output device. On input, n characters are read and inserted in to the Hollerith conversion field of the format specification. For example:

```
C  OUTPUT HEADING TO LINE PRINTER:
    WRITE (5,10) N
10  FORMAT(15H OUTPUT HEADING,I3)
C  INPUT HEADING FROM KEYBOARD FOR LATER USE; UP TO 30 CHARS:
    READ (6,20)
20  FORMAT(30H )
```

In line 10, the leading space in the line to be output is the carriage control character (see Section 8.14), the 14-character heading is output, followed by the integer value of N printed in a field of three characters. The comma following the Hollerith field is optional.

In line 20, the Hollerith field indicated is blank; however, any alphanumeric characters within this field would be replaced by

An alternative form is to enclose Hollerith data in single quotes. The result on both input and output is exactly as described for H conversion. A quote character within such a Hollerith string is represented by two successive single quotes. For example:

When output to the keyboard or line printer, the line:

is printed.

Spacing control within a record is performed through the use of the X specification to generate or skip a given number of characters or the T specification which allows the user to specify positions within a record.

WX

```
WRITE (5,75) I,X
75  FORMAT (5H STEPI5,10X2HY=F7.3)
```

$$\text{STEP}28 \text{ } Y = -3.872$$

8-13

A specification of the form:

Tw

specifies the character position in a FORTRAN record where the data transfer is to begin. The value of w must be an unsigned integer constant and indicates the wth character of the record on input and on non-printed output, and the (w-1)th character of a printed output record (where the first character is a carriage control character). For example:

```
2  FORMAT (T50,'BLACK'T30, 'WHITE')
```

would cause the following line to be printed:

<u>Print Position 29</u>	<u>Print Position 49</u>
↓	↓
WHITE	BLACK

The statements:

```
1  FORMAT (T35,'MONTH')  
   READ (3,1)
```

cause the first 34 characters of the input data to be skipped, and the next five characters would replace the characters M, O, N, T, and H in storage. If an input record containing:

ABC~~XXXX~~XYZ

is read with the format specification:

```
10 FORMAT (T7,A3,T1,A3)
```

then the characters XYZ and ABC are read, in that order.

8.14 CARRIAGE CONTROL

The first character of each ASCII record controls the vertical spacing of the line printer and teleprinter. When an output line (record) is generated, it is constructed within the system exactly as described by the combined WRITE and FORMAT statement designations. Before output to either of the printing devices, the first character

of the record is interpreted as the carriage control character and is deleted from the output line.

The carriage control characters are specified in Table 8-3.

Table 8-3
Carriage Control Characters

<u>Character</u>	<u>Effect</u>
blank	carriage return/line feed (advance printing position to beginning of next line)
Ø zero	carriage return/line feed/line feed (advance printing position to beginning of the line after the next)
1 one	carriage return/form feed (advance printing position to the beginning of the next line and perform a form feed where the hardware capability exists)
+ plus	carriage return (advance printing position to the beginning of the current line, allows current line, to be overprinted)
\$ dollar	inhibit carriage return/line feed (do not advance printing position, used when creating output lines to interact with the keyboard)

Any character other than those described in Table 8-3 which appears first in the line is deleted from the output line and treated as through it were a blank carriage control character.

8.15 REPETITION OF FIELDS AND GROUPS AND MULTIPLE RECORDS

A field specification can be repeated by preceding that specification with an unsigned integer denoting the number of repetitions. For example:

2Ø FORMAT(2E12.4,3I5)

which is equivalent to:

2Ø FORMAT(E12.4,E12.4,I5,I5,I5)

Similarly, a group of field specifications can be repeated by enclosing the group in parentheses and preceding the whole with an unsigned integer denoting the number of repetitions. For example:

```
3Ø FORMAT(2I8,2(E15.5,2F8.3))
```

which is equivalent to:

```
3Ø FORMAT(2I8,E15.5,F8.3,F8.3,E15.5,F8.3,F8.3)
```

Where several input/output records are specified in a single format specification, and those different records have different field specifications, a slash is used to indicate a new record. For example:

```
4Ø FORMAT(3Ø8/I6,2F8.4)
```

is equivalent to the two following FORMAT statements:

```
4Ø FORMAT(3Ø8)
41 FORMAT(I6,2F8.4)
```

A single record may be up to 133 characters long (this provides for a 132 character line printer line plus a carriage control character).

The separating comma can be omitted when a slash is used. The presence of n slashes at the beginning or end of a FORMAT statement cause n blank records to be output or to be skipped on input. When n slashes appear between the first and last FORMAT specification, n-1 blank records are output or skipped on input.

8.16 FORMATS STORED AS DATA

The ASCII character string comprising a format specification may be stored as the values of an array. Input/output statements may refer to the format by giving the array name, rather than the statement number of a FORMAT statement. The stored format has the same form as a FORMAT statement excluding the word "FORMAT." The enclosing parentheses are included.

As an example, consider the sequence:

```
      DIMENSION SKEL(2)
      READ 1, (SKEL(I), I=1,2)
1     FORMAT(2A4)
      READ(5,SKEL)K,X
```

The first READ statement enters the ASCII string into the array SKEL. In the second READ statement, SKEL is referenced as the format governing conversion of K and X.

Formats specified in an array need not be in .ASCIIZ format (see the MACRO-11 manual and Section 6.5.10); that is, such arrays need not have a zero stored in the last byte. For example, the format created below:

```
      REAL FORM1(3)
      DATA FORM1/'(I6','//I6','I3)'/
      WRITE(6,FORM1) I,J,K
```

is equivalent to:

```
      WRITE(6,50) I,J,K
50    FORMAT(I6 //I6,I3)
```

One caution should be noted with respect to the storing of formats within arrays. Such storage does not result in the correct linkage of the I/O conversion routines from the OTS Library. For example:

```
      REAL FORM1(2)
      INTEGER II
      LOGICAL LL
      DATA FORM1 / '(I5','L5)'/
      II=0
      LL=.TRUE.
      WRITE(6,FORM1) II,LL
      END
```

If the preceding program is run, it produces the run-time error message:

```
FORT008000 LINKAGE ERROR (MISSING FORMAT CONVERSION ROUTINE)
```

This is identical to the situation encountered in performing I/O in an overlay system (see discussion in Section 7.3). The remedy is to force the loading of the required I/O routines by using dummy FORTRAN I/O statements or an assembly language subprogram declaring the necessary global symbols. This technique is discussed in greater detail in Section 7.3.



CHAPTER 9

I/O AND DEVICE HANDLING STATEMENTS

9.1 INTRODUCTION TO FORTRAN I/O

Input of data to a FORTRAN program is performed via the READ statement. Output from a FORTRAN program is performed via either the WRITE or PRINT statement. These statements are generally used in conjunction with a FORMAT statement or format specification contained in an array. Each READ and WRITE statement contains a reference to the device to or from which the data transfer occurs. The PRINT statement assumes output to the line printer although the assumed device can be altered.

There are three categories of READ and WRITE statements:

- a. Formatted READ, WRITE and PRINT statements are used with or without an I/O list. Where a list of variables is provided, those values are transferred and automatically converted to ASCII for output or internal format on input. Where an I/O list is not provided, an alphanumeric data transfer is indicated.
- b. Unformatted READ or WRITE statements perform a transfer of binary information only. These statements are not used with FORMAT statements or specifications.
- c. Direct access disk READ and WRITE statements perform I/O of fixed-length records to a disk or DECTape file previously defined by a DEFINE FILE statement. Direct access READ and WRITE statements are not used with FORMAT statements or specifications.

A feature available with all forms of READ and WRITE statements allows the program to transfer control to a specified line number if an end-of-file or error condition develops during the I/O operation.

Table 9-1 shows the skeleton formats for each of the three categories described above.

The remaining statements covered in this Chapter perform file-oriented or device-oriented operations with respect to bulk storage devices such as magnetic tape, disk and DECTape.

Table 9-1
READ and WRITE Statement Summary

Category	Input Statements	Output Statements
formatted	$\left\{ \begin{array}{l} \text{READ (u,f) list} \\ \text{READ (u,f)} \\ \text{READ f, list} \end{array} \right.$	$\left\{ \begin{array}{l} \text{WRITE (u,f) list} \\ \text{WRITE (u,f)} \\ \text{PRINT f, list} \end{array} \right.$
unformatted	$\left\{ \begin{array}{l} \text{READ (u) list} \\ \text{READ (u)} \end{array} \right.$	$\left\{ \begin{array}{l} \text{WRITE (u) list} \\ \text{WRITE (u)} \end{array} \right.$
direct disk access	$\left\{ \begin{array}{l} \text{READ (u'r) list} \\ \text{READ (u,f,END=n) list} \\ \text{READ (u,f,ERR=n) list} \\ \text{READ (u,f,END=n,ERR=n) list} \end{array} \right.$	$\left\{ \begin{array}{l} \text{WRITE (u'r) list} \\ \text{WRITE (u,f,END=n) list} \\ \text{WRITE (u,f,ERR=n) list} \\ \text{WRITE (u,f,END=n,ERR=n) list} \end{array} \right.$
transfer on END= and ERR= can also be used with unformatted and direct access I/O statements		
Where: - u is a logical unit number, either an integer constant or variable - f is a format reference (line number of a FORMAT statement or an array name) - list is a list of I/O variables - r is an integer variable or constant designating the record number - n is a statement number ' is the character used to separate the symbolic file number and the record number , is the character used to separate all other arguments of I/O statements *For example: READ(u,END=n) list WRITE(u'r,ERR=n) list		

9.1.1 I/O Devices

PDP-11 FORTRAN supports all I/O devices supported by the DOS Monitor. The default FORTRAN logical device assignments are described in Table 9-2. Device assignments can be altered by means of the SETFIL subroutine (Section 6.5.3), by means of the Monitor ASSIGN command, or by reassembling the FORTRAN Device Table (see Section 10.5).

Table 9-2

FORTRAN Logical Device Assignments

Logical Unit Number	Device
1	system disk, SY:
2	system disk, SY:
3	system disk, SY:
4	high-speed paper tape reader, PR:
5	line printer, LP:
6	terminal, KB:
7	system disk, SY:
8	BATCH system input device, BI:

The error logging device has logical dataset name CMO and is defaulted to the terminal keyboard. The error logging device is used for run-time error messages and PDUMP output.

9.1.2 I/O Records

All data is transmitted by an Input/Output statement in terms of records. The maximum amount of information in one record and the manner of separation between records depends upon the medium. For punched cards, each card constitutes one record; on a teletypewriter a record is one line; for ASCII records, the amount of information is specified by the FORMAT reference and the I/O list; for unformatted binary records, the amount of information is specified by the I/O list; DEFINE FILE is used to specify record length for direct-access files.

Each execution of an input or output statement initiates the transmission of a new data record. Thus, the statement:

```
READ 2,FIRST,SECOND,THIRD
```

is not necessarily equivalent to the statements:

```
READ 2,FIRST  
READ 2,SECOND  
READ 2,THIRD
```

In the second case, at least three separate records are required, whereas, the single statement:

```
READ 2,FIRST,SECOND,THIRD
```

may require one, two, three, or more records depending upon FORMAT statement 2.

If an Input/Output statement requests less than a full record of information, the unrequested part of the record is lost and cannot be recovered by another Input/Output statement without repositioning the record. Repositioning is not possible on all devices (see Sections 9.6 and 10.1). If an input list requires more than one ASCII record of information, successive records are read.

9.1.3 I/O Lists

An Input/Output list contains the names of variables and array elements whose values are assigned on input or written on output. Input and output lists have the same form, consisting of individual elements separated by commas.

The simplest form of I/O list consists of a list of variable names or array element names. For example:

```
A,B,C
```

indicates three simple variable names.

An array name used alone in an I/O list specifies all of the array elements defined when the array is declared. Elements are read into core or output in the order specified for array storage (see Section 5.3.1). Thus, where the maximum element of the array X is X(400), an I/O list to include the entire array X could be written:

A,B,C,X

The elements of a list are specified in the order of their occurrence from left to right and may be enclosed in parentheses. The following list is identical in meaning to the previous list:

(A,B,C),X

An implied DO loop within the I/O list can be used to indicate transmission of only part of an array. This is performed by including a parenthesized quantity as a list element as follows:

One-dimensional Array

(A(I),I=n,m,k)

Two-dimensional Array

((A(I,J),I=n,m,k),J=n',m',k')

or

(A(I),I=n,m)

((A(I,J),I=n,m),J=n',m')

where A is an array name, I and J are array subscripts, and n,m, and k represent the initial, terminal and step values of the implied loop. For example:

(X(K),K=1,4)

is equivalent to:

X(1),X(2),X(3),X(4)

The implied DO loop:

((Y(I,J),I=1,2),J=1,3)

is equivalent to:

Y(1,1),Y(2,1),Y(1,2),Y(2,2),Y(1,3),Y(2,3)

As shown above, nesting is used to provide for various levels of indexing, up to three dimensions.

9.2 FORMATTED I/O

9.2.1 Formatted READ Statement

A formatted READ statement is of the form:

```
READ (u,f) list  
READ (u,f)
```

or

```
READ f, list      [equivalent to READ (4,f)]
```

where u indicates a logical unit number and f indicates a format reference (either a FORMAT statement or an array name whose elements contain a format specification).

A formatted READ statement causes information to be read from the specified device and placed in memory. The data is converted from external to internal format as indicated by the format specification. If an I/O list is provided, data items are stored as the values of the listed variables. If a list is not provided the data is stored as an alphanumeric string in the format itself. For example:

```
READ (1,100) A,B,C
```

reads three values from device 1 according to the format in the FORMAT statement at line 100 and assigns the values to the three variables named.

```
      READ (1,100)  
100  FORMAT (5HHELLO)
```

When the above READ statement is executed, and the characters ABCDE (for example) are given as input, the FORMAT statement at line 100 is now

```
100  FORMAT (5HABCDE)
```

A READ statement of the form:

```
READ 110,X,Y,Z
```

is similar to the first form discussed, except that input is assumed

to come from logical unit 4 (default to PR:). The FORMAT statement associated with the operation is at line 110.

```

          DIMENSION B(32)
50        READ (6,2000) B
          .
          .
100       READ (6,B) I,J,K,L
          .
          .
200       FORMAT (32A4)
```

When statement 50 is executed, the value of the array B is read from device 6. If the first two elements of B contain the following:

```
B(1) = '(4I6'
B(2) = ').'
```

and the remaining elements of B are blank as a result of the first read. (The FORMAT statement in line 200 provides formatting information for the user to input enough data to completely fill the array B.) When statement 100 is executed, the contents of the array B becomes the format reference for that READ statement. The variables I, J, K and L are read from device 6 according to the format 4I6 (that is, four 6-digit integers). This type of READ statement (statement 100) is said to contain variable read-in formats and allows the user to specify an I/O format at run time (see Section 8.16).

Any format conversions and I/O control routines not needed in the resident section but required by overlay sections must be forcibly loaded into the resident section. This can be done by declaring the appropriate globals in an assembly language routine or inserting dummy FORMAT and Input/Output statements in the resident main program for all those routines needed in the overlays and not required in the resident section. See Section 7.3 for further details.

9.2.2 Formatted WRITE Statement

A formatted WRITE statement is of the form:

```
WRITE (u,f) list
```

or

```
WRITE (u,f)
```

where u indicates a logical unit number and f indicates a format reference.

A formatted WRITE statement causes one or more logical records to be constructed and written on the designated device in ASCII format. A logical record contains up to 133 characters, including the carriage control character; the record format is described more completely in Section 10.2.1. The data is converted to external form as specified in the designated format reference. If an I/O list is provided, the values of the variables indicated are output. If a list is not provided, the information is read directly from the format reference and written on the device designated in ASCII form (this is generally done with Hollerith output). For example:

```
      WRITE (5,115) IERR,INET,ILOW
115  FORMAT (3I4)
```

The above combination outputs three integer values onto device 5.

```
      WRITE (5,120)
120  FORMAT (25H THIS IS HOLLERITH OUTPUT)
```

The above combination outputs 25 characters to form the message:

```
      THIS IS HOLLERITH OUTPUT
```

on device 5.

9.2.3 PRINT Statement

The PRINT statement is of the form:

```
      PRINT f, list
```

where f indicates a format reference. The PRINT statement is a formatted WRITE statement whose output is always sent to the same device, the line printer. PRINT performs a WRITE (5,f). The default device can be changed by using the SETFIL statement or ASSIGN Monitor command with respect to logical unit 5.

As an example:

```
PRINT 115, NA,NB,NC
```

This outputs three values according to the format described in line 115. This statement is analogous to the third type of READ statement described in Section 9.2.1.

9.3 UNFORMATTED I/O

9.3.1 Unformatted READ Statement

An unformatted READ statement is of the form:

```
READ (u) list
```

or

```
READ (u)
```

where u indicates a logical unit number.

An unformatted READ statement causes binary information to be read from the unit designated and stored in memory as the values of variables in the I/O list, if any. For example:

```
READ (7) A,B,C
```

This statement reads enough binary values from device 7 to fill three real variables (2-words each). If A, B, and C have been previously defined as double-precision variables, then three real values (4-words each) are read, and so on.

```
READ (7)
```

allows the user to skip over one record on device 7. One such statement must appear for each record to be skipped.

Each READ statement reads one unformatted binary record. If the I/O list does not use all values in the record, the remaining values are discarded. If more values are required than are contained in the record, a run-time error message results.

Unformatted binary I/O is generally used to enable the storage of intermediate results from one program to another without the loss of precision involved in formatting data output. Unformatted I/O is performed between memory and such peripherals as paper tape, DECTape, magnetic tape or disk, but not to an ASCII device where such I/O is generally meaningless.

9.3.2 Unformatted WRITE Statement

An unformatted WRITE statement is of the form:

```
WRITE (u) list  
or  
WRITE (u)
```

where u indicates a logical unit number.

An unformatted WRITE statement causes binary information to be read from memory and written on the device indicated in binary form. For example:

```
WRITE (7) A,B,C
```

This statement writes the binary values of the variables A, B and C onto device 7.

```
WRITE (7)
```

allows the user to write one null record on device 7. One such statement must appear for each null record.

9.4 DIRECT ACCESS I/O

Direct access I/O can be performed on any directory structured device.

9.4.1 DEFINE FILE Statement

The DEFINE FILE statement defines the record structure of a disk or DECTape file. File space is allocated by means of a call to the SETFIL subroutine or by the DEFINE FILE statement if no file already exists. However, DEFINE FILE automatically allocates a file if none exists. The DEFINE FILE statement can be used as either a declarative (non-executable) or executable statement in

the program. Where DEFINE FILE is executable, arguments can be computed within the program.

The DEFINE FILE statement is formatted as follows:

```
DEFINE FILE u(m,n,U,ivar)
```

where the arguments are as follows:

- u is an integer constant or variable used as the logical unit number for this file specification.
- m is an integer constant or variable defining the number of records in the disk file. The records in the file are numbered from 1 to m.
- n is an integer constant or variable defining the length (in words) of each file record.
- U is a fixed argument designating that the disk file is unformatted (i.e., binary). No other characters are legal.
- ivar is an integer variable, called the associated variable, which is set to point to the next record at the conclusion of an I/O operation on the file. If the associated variable is to be used by more than one subprogram or in an overlay environment, v should be placed in a resident common block.

The statement:

```
DEFINE FILE 1(1000,100,U,IVAR)
```

specifies a 1000-record file, each record of which is 100 words long.

The variable IVAR maintains an index of records processed, providing a pointer to the next record.

With respect to overlays, the DEFINE FILE routine uses storage space which must be allotted such that the overlay system is not corrupted. In the statement:

```
DEFINE FILE 1(10,10,U,INDX)
```

the variable INDX (the associated variable) is used to point to the record to be processed for statements such as:

```
READ (1'1)  
WRITE(1'1)  
FIND (1'1)
```

Every READ, WRITE or FIND done on unit 1 modifies INDX. INDX must be placed in the local storage of the resident section or in a resident common block. If it is not, the FORTRAN I/O system may destroy an unpredictable location in core when attempting to store into INDX.

9.4.2 FIND Statement

The format of the FIND statement is:

```
FIND(u'r)
```

where u is the file specification as assigned by the DEFINE FILE statement, and r is an integer constant or variable specifying a record within the file a. The FIND statement is included for compatibility with other FORTRAN systems. The DOS Monitor does not support disk head positioning; therefore, the statement is ignored during compilation and execution.

9.4.3 Direct Access READ Statement

A direct access READ statement is of the form:

```
READ (u'r) list
```

where u is a logical unit number as established in a prior DEFINE FILE statement and r is an integer variable or constant specifying the record number. The list specifies the number of items to be read. An apostrophe is used to separate the two elements to differentiate this form of READ statement from a formatted READ statement.

A direct access READ statement provides random access to fixed-length records in a disk or DECTape file. For example:

```
READ (1'50) ADC, ADX, AOD
```

record number 50 of the file defined as file 1 is read to obtain values for the variables ADC, ADX and AOD. Three 2-word values are transmitted since the variables indicated are real variables. If the

variables were byte, integer, complex, or double-precision, a corresponding number of bytes or words of data would be read for each variable.

As in other I/O operations, only the number of items in the list are transmitted regardless of the number of items in the record. If a list requests more values than are provided in the record, an error message is printed.

Implied DO loops can be included in direct access I/O statements.

9.4.4 Direct Access WRITE Statement

A direct access WRITE statement is of the form:

```
WRITE (u'r) list
```

where a is a logical unit number and r is an integer variable or constant specifying the record number. The list specifies the number of items to be output. An apostrophe is used to separate the two elements to differentiate this form of WRITE statement from a formatted WRITE statement.

The direct access WRITE statement outputs a fixed-length record directed into a file and is analogous to the direct access READ statement described in the previous Section.

9.5 TRANSFER OF CONTROL ON ERROR CONDITION

Both READ and WRITE statements can be written so as to transfer control to a given statement number if an error condition is detected. These variations can be specified in one of the following formats:

```
READ(u,f,END=n) list
READ(u,f,ERR=n) list
READ(u,f,END=n,ERR=n) list
WRITE(u,f,END=n) list
WRITE(u,f,ERR=n) list
WRITE(u,f,END=n,ERR=n) list
```

where u is a logical unit designation, f is a format reference, and n is a line number within the program. The list is optional. END= and ERR= can also be used with unformatted and direct-access I/O.

The arguments END=n and ERR=n can appear separately or together (in any order). If an end-of-file condition is encountered during a READ, control transfers to the statement with line number n associated with the END parameter. If an END parameter is not specified, I/O on that device terminates and the program terminates with an error indication.

If an error on input or output is encountered, control transfers to the statement with line number n associated with the ERR parameter. If an ERR parameter is not specified, the program terminates with an error message.

9.6 DEVICE CONTROL STATEMENTS

Table 10-1 defines the types of I/O and specific operations which can be performed on each of the Standard DOS peripherals.

9.6.1 BACKSPACE Statement

The BACKSPACE statement is of the form:

BACKSPACE u

where u is an unsigned integer constant or integer variable specifying the logical unit number of a device. The unit is repositioned to the beginning of the file opened on that unit, and the read/write mechanism spaces forward n-1 records, where n is the number of the record processed before the BACKSPACE. The effect is to backspace one ASCII record or one logical binary record, depending upon the device.

9.6.2 REWIND Statement

The REWIND statement is of the form:

REWIND u

where u is an unsigned integer constant or integer variable specifying the logical unit number of a device. This statement repositions the designated unit to the beginning of the currently opened file.

9.6.3 END FILE Statement

The END FILE statement is of the form:

```
END FILE u
```

where u is an unsigned integer constant or integer variable specifying the logical unit number of a device on which input or output operations are being performed. The END FILE statement uses the DOS Monitor .CLOSE. operation to close the file. END FILE also closes any open random-access files to allow another DEFINE FILE to be performed.

9.7 ENCODE and DECODE

ENCODE and DECODE statements transfer data, according to format specifications, from one section of memory to another. DECODE changes data in ASCII format to the specified format. ENCODE changes data of the specified format into ASCII format. The two statements are of the form:

```
ENCODE (c,f,v)list
DECODE (c,f,v)list
```

where the arguments are as follows:

c	the number of characters (bytes) to be converted
f	the FORMAT statement number or array specifying the format
v	the name of the <u>array</u> containing the ASCII character string, must be an array name.
list	the list of variables to be changed from or into ASCII format, and is the same as any other I/O list (see Section 9.1.3).

As an example of the use of the DECODE statement:

```
DIMENSION A(3),I(3)
DATA A/'1234','5678','9012'/
DECODE (12,1000,A)I
1000 FORMAT (3I4)
```

The previous code causes the integer array I to take on the numeric values of the three strings stored in the ASCII array A, as follows:

```
I(1) = 1234
I(2) = 5678
I(3) = 9012
```

```

        BYTE BUFF(80),TITLE(12)
C
C   GET NEXT RECORD
5   READ(1,10,END=900) BUFF
10  FORMAT(80A1)
C
C   DECODE TYPE FIELD AND BRANCH ON IT
C
        DECODE(80,20,BUFF) ITYP
20  FORMAT(I2)
        IF (ITYP.GE.1.AND.ITYP.LE.3) GOTO (100,200,300) ITYP
C
C   BAD RECORD
C   WRITE(5,30)BUFF
30  FORMAT (' **BAD RECORD**'/1X,80A1)
        GO TO 5
C
C   ITYP=1
100 DECODE(80,110,BUFF) I,J,K
110 FORMAT(T4,2I4,T24,I6)
        .
        .
        .
C
C   ITYP=2
200 DECODE(80,210,BUFF) A,B,C,D,I,J
210 FORMAT(T20,4F10.5,2I5)
        .
        .
        .
C
C   ITYP=3
300 DECODE(80,310,BUFF) TITLE,I,J
310 FORMAT(T3,12A1,T60,2I5)
        .
        .
        .
C   END-OF-FILE TRANSFERS TO 900
900 CONTINUE
        .
        .
        .

```

Figure 9-1
DECODE Example

DECODE is analogous to a READ statement, since it causes conversion from ASCII to internal format. ENCODE is analogous to a WRITE statement, causing conversion from internal data formats to ASCII.

The DECODE statement is used when it is necessary to process records having different formats. Figure 9-1 contains a segment of FORTRAN code which reads 80-character records from logical unit 1. The first two characters of each record contain an integer between 1 and 3 indicating the format of the remainder of the record. DECODE is used twice in the example: first to scan the record type field and then to read and translate the remainder of the record.



2

2



2

2



CHAPTER 10

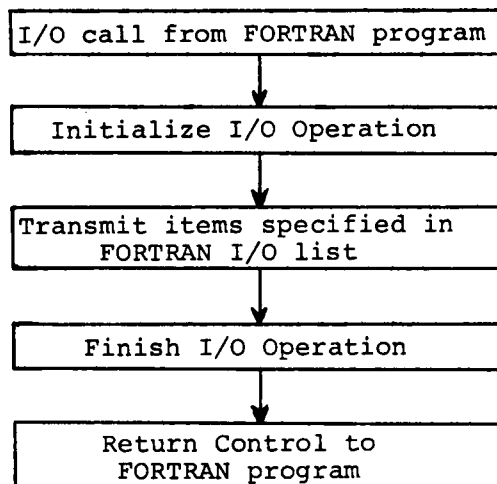
FORTRAN/DOS I/O CONSIDERATIONS

Input/output operations for a FORTRAN-compiled program are performed by the FORTRAN Object Time System. The FORTRAN language statements related to I/O are described in Chapters 8 and 9. It is the purpose of this Chapter to describe the processing of I/O requests in greater detail, with emphasis on the FORTRAN OTS and DOS Monitor mechanisms and representations. File formats and interactions between DOS and the FORTRAN OTS are described.

The FORTRAN OTS accomplishes all physical I/O through the DOS Monitor. FORTRAN I/O is, thus, device-independent to the extent that device-independence is supported by the DOS Monitor. For example, FORTRAN formatted WRITE statements can be directed to any DOS ASCII output device, but direct-access I/O can only be done on devices having fixed-length addressable blocks (disks and DECTape, but not industry-standard magnetic tape).

10.1 INPUT/OUTPUT OVERVIEW

As described in Chapter 9, all FORTRAN I/O operations are done in terms of logical records. Each READ or WRITE statement transmits one or more logical records. The overall flow of control in processing a single FORTRAN READ or WRITE statement is:



The operations accomplished are as follows:

a. Initialize I/O Operation:

1. Verify legal unit number, obtain \$DEVTB entry.
2. Test to see that the file is open; if it is not, open the file.
3. If the file is open, check for compatibility of device and file for type of I/O requested.

If a contiguous file is to be allocated, this is done as a part of the file-opening process in step 2 above.

b. Item Transmission:

1. If the operation is a READ, obtain a record if needed; check for I/O errors.
2. Transmit items according to I/O list.
If record length is exceeded, issue a diagnostic message.

c. Finish I/O Operation:

1. Complete current record with padding and record separators as required.
2. If the operation is a WRITE, transmit record and wait for completion.
3. Evaluate physical I/O error returns.

In the above descriptions, a file is not opened until a READ or WRITE statement for that unit is executed. Thus, errors such as non-existent files or inability to allocate a contiguous file are not detected until an I/O operation is attempted.

All FORTRAN I/O is single-buffered. A DOS .WAIT macro is issued after each Monitor I/O call. All transfers of control on END= and ERR = precisely identify the FORTRAN I/O statement causing the END= or ERR= transfer.

The principal data structure used by the FORTRAN OTS for controlling I/O is the device table, \$DEVTB, described in Sections 10.4 and 10.5. The unit number supplied in a READ or WRITE statement is used to select a device table entry. The device table entry contains information for each logical unit, including:

Filename
Physical device specification and unit number
User identification code (UIC)
Protection code
Open/close status

FORTTRAN logical unit defaults are specified by the values assembled into \$DEV TB in the version supplied with the FORTTRAN OTS. CALL SETFIL (Section 6.5.3) can be used to modify \$DEV TB entries under program control. The DOS ASSIGN command can be used to override some of the \$DEV TB attributes.

The physical devices supported by FORTTRAN and DOS, and the types of FORTTRAN operations permitted on each device, are specified in Table 10-1.

10.2 FILE STRUCTURES AND FORMATS

FORTTRAN input/output facilities are provided by three packages of OTS routines: formatted I/O, unformatted I/O, and direct-access I/O. General characteristics of DOS file structures and the three I/O packages are discussed in following Sections.

The DOS Monitor provides a variety of I/O modes, and two principal file structure organizations. The file structures are:

- a. Linked files for sequential access, and
- b. Contiguous files for direct or sequential access.

Linked files consist of a series of blocks which need not be physically contiguous on the device. Each block contains a pointer to the next block of the file. When a linked file is initially opened for output, the first block is allocated. As output requests occur, additional blocks are allocated and linked to the file. The length of the file is, therefore, not known at the time the file is opened.

Formatted and unformatted output ordinarily use linked files. A contiguous file may be used for output following a file allocation operation through SETFIL.

Table 10-1

PDP-11 FORTRAN IV Standard Peripheral Devices

Name	Device Specification	FORTRAN I/O Type		FORTRAN I/O Operations	
		Formatted Unformatted Direct-access		READ WRITE ENDFILE BACKSPACE REWIND	
Disk*	DC: } or SY:				
	DF:	X	X	X	X
	DK:				
DEctape	DTn:	X	X	X	X
Magtape	MTn:	X	X	X	X
Line Printer	LP:	X		X	
Card Reader	CR:	X	X	X	
Terminal	KB:	X		X	
High-speed paper tape reader	PR:	X	X	X	
High-speed paper tape punch	PP:	X	X	X	
Low-speed paper tape reader-punch	PT:	X		X	

* the universal mnemonic SY can be used to specify the system disk independent of whether that unit is an RC11, RF11 or RK11 disk.

X indicates that the I/O mode or operation is supported on the indicated device.

Contiguous files are specifically intended for direct-access I/O. A fixed-size contiguous file must be initially allocated on disk or DECTape by means of DEFINE FILE or CALL SETFIL. Direct-access I/O statements may then read or write individual records without having to read all preceding records.

Table 10-2 summarizes the file structures and DOS I/O modes used by the FORTRAN I/O packages.

Table 10-2
File Structures and I/O Modes

FORTRAN I/O Type	DOS File Structure	DOS I/O Mode
Formatted Unformatted Direct-access	Linked or Contiguous Linked or Contiguous Contiguous	Formatted ASCII Normal Formatted Binary Normal Unformatted Binary

10.2.1 FORTRAN Formatted I/O

The formatted input/output routines read or write variable-length formatted ASCII records. A record consists of a maximum of 133 ASCII characters transmitted under control of a format specification, followed by record separator characters (typically carriage return/line feed). A maximum length record is transmitted by 133A1.

On output to a non-printing device (e.g. disk file), the record separator consists of the carriage return/line feed sequence. The maximum length output line thus requires 135 bytes on a disk file. The file consists simply of a sequence of output lines; there are no additional headers, checksums, or byte counts.

On output to a printing device (KB: or LP:) the record separator appended to each record consists of a carriage return and a vertical tab. The first character of each record is deleted from the record and is interpreted as a carriage control character, and may cause output of 0, 1 or 2 line feed characters or a form feed character.

On input, records longer than 135 bytes (including separator) are truncated. For shorter records, the last character (line feed, form feed or vertical tab) is deleted. The next-to-last character is deleted if it is a carriage return. The record is then padded with blanks to 133 characters. No other special interpretations are provided; that is, form feeds, tabs and other special ASCII characters are passed to the program as single characters. In particular, tab characters are not converted to blanks.

Note that interpretation of the carriage control character is provided by the FORTRAN OTS, not by the DOS Monitor or utility programs. If output containing FORTRAN carriage control characters is directed to a disk or tape file for later printing, the printing must be done under control of a FORTRAN program in order to obtain the desired carriage control.

10.2.2 FORTRAN Unformatted I/O

The unformatted input/output routines read or write variable length binary records using the DOS formatted binary mode. Each record contains checksums for parity checking and a byte count.

A FORTRAN unformatted WRITE statement outputs a single logical record. A logical record consists of one or more segments, each segment containing a maximum of 61 words of data from the user program and a segment control word supplied by the FORTRAN OTS. The segment control word is inserted as the first data word of the segment and may take one of the following values:

Table 10-3

Segment Control Word

Value	Meaning
Ø	Not first or last segment
1	First segment of logical record
2	Last segment of logical record
3	First and last segment of logical record

Each segment of the logical record is output as a DOS formatted binary line. The format of each segment is shown in Figure 10-1.

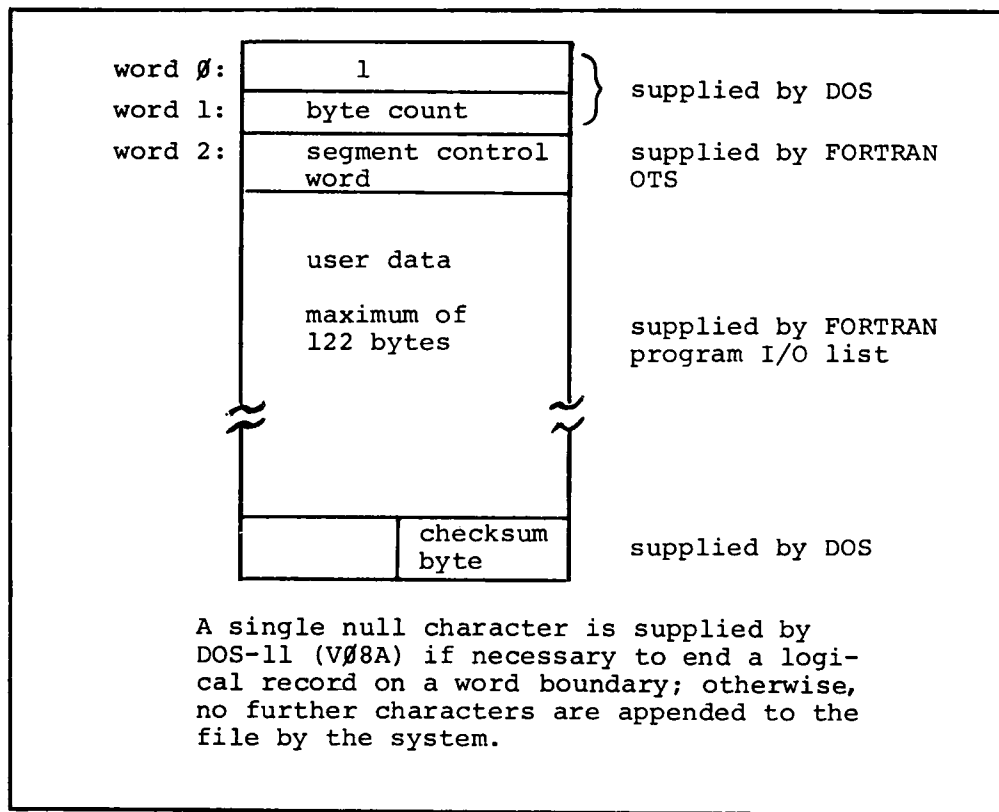


Figure 10-1

Logical Record Segment Format

Each segment has four words of overhead; the user must allow for these overhead words when allocating a contiguous file for unformatted I/O through CALL SETFIL.

10.2.3 FORTRAN Direct-Access I/O

The direct-access I/O routines read or write fixed-length binary records in a contiguous file. The DOS unformatted binary I/O mode is used. The maximum record length is 32767 bytes.

The logical record structure for a direct-access file is determined entirely by the DEFINE FILE statement. The records themselves contain no checksums, byte counts, or record separators. Different programs can process the same file using different logical record structures. In Figure 10-2, Program 1 operates on records consisting of 10 x 10 integer matrices, while Program 2 operates on records consisting of a single row of the matrix:

```

Program 1
  INTEGER MATRIX(10,10)
  .
  .
  .
  DEFINE FILE 1 (20, 100, U, INDX)
  .
  .
  C WRITE OUT A 10 x 10 MATRIX
  WRITE (1'NREC')((MATRIX(I,J), J=1,10), I=1,10)
  .
  .
  .
  END

Program 2
  INTEGER ROW(10)
  .
  .
  .
  DEFINE FILE 1(200, 10, U, INDEX)
  .
  .
  .
  C READ A 10 ELEMENT ROW OF MATRIX
  READ(1'NREC') ROW
  .
  .
  .
  END

```

Figure 10-2

Program Example Using Logical Records

If a WRITE statement transmits fewer words than will fit in the record, the remainder of the record is filled with zeroes.

The direct-access record structure is independent of the physical block size of the I/O device, but more efficient operation results if the record size is an exact multiple of the physical block size (RF and RC disks = 64 words, RK and DECTape = 256 words).

10.3 FORTRAN I/O ERROR HANDLING

There are three classes of run-time error conditions specifically associated with FORTRAN I/O operations.

Class 1 - Non-existent file, physical I/O error, etc.

Class 2 - FORMAT syntax errors

Class 6 - Conversion errors during format conversions

A complete list of FORTRAN run-time errors is contained in Appendix E.

As described in Chapter 9, READ and WRITE statements can specify transfer of control on an end-of-file (END=) and on certain physical I/O conditions (ERR=). The SETFIL subroutine (Section 6.5.3) can specify an error variable, IERR, which is set to indicate the specific type of I/O error.

Errors in class 2 and class 6 are processed as described in Section 11.5 and Appendix E. Class 1 I/O error conditions are processed as follows:

Assume the error condition is identified by the diagnostic FORT0010NN.

1. Set the IERR variable, if specified in the previous SETFIL, to NN.
2. If END=n is specified and the error condition is an end-of-file, (FORT001004), transfer control to statement n.
3. If ERR=n is specified, transfer control to statement n.
4. Otherwise, process error according to Section 11.5.

10.4 FORTRAN DEVICE ASSIGNMENTS

The default FORTRAN device assignments are shown in Table 10-4. These assignments can be temporarily altered through use of a call within the user program to the SETFIL routine or by use of the ASSIGN Monitor command. Section 10.5 describes how to permanently alter the FORTRAN device assignments.

Table 10-4

FORTRAN Logical Device Assignments

Logical Unit Number	Device	Default File Name
1	system disk, SY:	FORT001.DAT
2	system disk, SY:	FORT002.DAT
3	system disk, SY:	FORT003.DAT
4	high-speed paper tape reader, PR:	FORT004.DAT
5	line printer, LP:	FORT005.DAT
6	terminal keyboard, KB:	FORT006.DAT
7	system disk, SY:	FORT007.DAT
8	BATCH system input device, BI:	FORT008.DAT

Device assignments are determined in one of three ways. These are described below in order of their increasing priority.

- a. The device table default values described in Table 10-4.
- b. A call within a FORTRAN program to the SETFIL subroutine (see Section 6.5.3) overrides the default device assignments.
- c. The Monitor ASSIGN command overrides both default assignments and any assignments made within the program by a call to the SETFIL routine.

The SETFIL routine causes entries to be made in the FORTRAN device table, \$DEVTB, which is described in greater detail in Section 10.5.

An ASSIGN Monitor command for use with FORTRAN is specified in the form:

```
$ASSIGN dev:file.ext[uic], n
```

where the \$ character is printed by the system, dev:file.ext[uic] represents a file specification, and n represents a FORTRAN unit number.

For example:

```
$ASSIGN DT:,2
```

forces a search on DECTape unit 2 for the filename (and extension) associated in the device table (or in a SETFIL call) with logical unit 2.

```
$ASSIGN DT3:FILE.TMP,3
```

causes the file FILE.TMP on DECTape unit 3 to be associated with logical unit 3.

Those alterations in the device table made by a call to the SETFIL subroutine are only valid during the program in which they appear. Device assignments made with the DOS ASSIGN command vary in scope depending upon when the command is given.

- a. When no program is in core, assignments made through an ASSIGN command are valid through several runs of the same or different programs and can be removed by an ASSIGN command with no arguments. For example:

```
$KI  
$AS DT:,1  
$RU X  
$RU Y
```

During programs X and Y, DECTape unit 0 is treated as logical unit 1.

- b. When a program is in core, assignments made through an ASSIGN command are valid only through the running of that program. For example:

```
$GE X
$AS DT:,1
.
.
$KI
```

During the run of program X, DECTape unit 0 is treated as logical unit 1. The KILL command returns the device assignments to their original default conditions.

- c. When a program is in core and executing, an assignment can be made through an ASSIGN command which is valid only through the running of that program. For example:

```
$RU X
PLEASE ASSIGN INPUT ON UNIT 4
A005 100 (PAUSE message)
$AS SY:FILE1.XX,4
$CO
```

During the run of program X, the FORTRAN program wrote a message requesting a device assignment and then executed a PAUSE statement. The user then indicated that the desired file was to be read from the system disk. Following the execution of program X, the device table reverts to its default assignments.

10.5 THE FORTRAN DEVICE TABLE, \$DEVTB

Logical device assignment is governed by the device table, which contains entries for 8 devices. Additional entries can be inserted, default entries changed, or entries deleted by reassembling the device table file. In order to allow for such changes, the source file for the device table (DVB09.PAL) is supplied with each FORTRAN system. When the source has been altered and assembled, the new device table object module can replace the supplied module in the FORTRAN Library or be linked as an object module (to individual programs) preceding the Library file. The device table consists of a 12-word header followed by a 16-word entry for each device.

Table 10-5
Device Table Entry

<u>Header</u>		
	Word 0	Address of device table entry for logical unit -3 (the message logging unit)
	Word 1	Address of entry for error routine message file
\$DEVTB:	Word 2	Number of entries in device vector table
	Word 3	Device number of message logging file
	Words 4-11	Addresses of device table entries for each of the devices 1 through 8.
<u>Entry</u>		
	Word 1	Link pointer (from Link Block, after .INIT)
	Word 2	Physical device name stored in Radix-50 format.
	Word 3	Bits 15-8, unit number (Default 0); Bits 7-0, DOS Open Code, taken from File Block-2
	Words 4-5	Filename (stored as .RAD50/FOR/,/NNN/;NNN=Entry Number)
	Word 6	Filename extension (stored as .RAD50/DAT/)
	Word 7	Switches* and protection code (Default=233)
	Word 8	Status/Mode (from Line Buffer Header)
	Word 9	Count of I/O operations for this device
	Words 10-14	Unused for formatted and unformatted I/O
	Word 15	User Identification Code (UIC) (default=0, which indicates the current UIC)
	Word 16	Address of error variable (IERR from CALL SETFIL)
<u>For Random I/O, Words 8-14 are as follows:</u>		
	Word 8	Function word
	Word 9	Block number
	Word 10	Buffer address
	Word 11	Buffer length
	Word 12	Associated variable address (from DEFINE FILE)
	Word 13	Maximum number of records (from DEFINE FILE)
	Word 14	Record length (from DEFINE FILE)
<u>*Switches are as follows:</u>		
<u>Bit</u>	<u>Setting</u>	<u>Meaning</u>
0-1	0	Closed
	1	Open formatted
	2	Open unformatted
	3	Open random

Table 10-5 describes the device table header and individual entry formats. The supplied device table is listed in Appendix F.

In order to add a device to the table:

- a. Alter the word at \$DEVTB to reflect the number of entries in the device table (the number of devices available to FORTRAN). The supplied value at \$DEVTB is 8.
- b. Append the address of the device entry to the 8-word header sequence of device table entry addresses (known as the device table entry vector).
- c. Append the new 16-word entry describing the additional device.

To delete a device from the table:

- a. Alter the word at \$DEVTB to reflect the entries in the device table.
- b. Set the address of the device entry in the device table entry vector to zero.
- c. Delete the entry for that device from the device table.

The default physical device for a table entry can be changed by modifying the second word of the entry. This allows the user to alter the associations between devices and logical unit numbers in Table 10-4.

The default filename and extension for a table entry can be changed by modifying words four through six of the table entry.

10.6 I/O EXAMPLE

This Section presents an example using the FORTRAN I/O system. The program FORDGN.FTN on the following pages is used in the FORTRAN system to read an ASCII source file and produce a contiguous file of specially formatted records containing the English language diagnostic messages for the FORTRAN Compiler and OTS. A listing of the message file is also printed on the line printer in the example following.

The program reads an input file from logical unit 4. The first record of the input file specifies the number of error messages in the output file and the name of the output file. CALL SETFIL sets the name of the output file and the DEFINE FILE statement specifies the record structure.

The source listing of FORDGN.FTN follows:

```

C      FORDGN.FTN
C      *****
C      CREATES AND THEN PRINTS A CONTIGUOUS FILE
C      OF MESSAGES FOR ACCESS BY THE FORTRAN COMPILER OR OTS.
C      *****
C      INPUT FORMAT (FROM LOGICAL UNIT 4):
C      1ST RECORD-- I3,10A1  I3=#OF 64 CHARACTER MESSAGES
C                      TO BE ALLOCATED. 10A1=FILE NAME
C      OTHER RECORDS- I3,32A2  I3=POSITION OF CURRENT
C                      MESSAGE IN THIS FILE. 32A2=CURRENT MESSAGE.
C      LAST RECORD-- I3,32A2  I3=NEGATIVE INTEGER. 32A2=IGNORED.
C      OUTPUT FILE IS CONTIGUOUS -- SY2:FILNAM[LOGIN=UIC]
C      PRINTED OUTPUT GOES TO LP: -- FORTRAN DEVICE 5.
C
      INTEGER A(32),NULL(32),COUNT,INDEX
      BYTE TODAY(9),FILNAM(10)
      DATA NULL / 32 * ' ' /
      COUNT=0
      WRITE (6,1000)
1000  FORMAT(' FORDGN V302A1/' INPUT FROM LOGICAL UNIT 4'/)
      ENDFILE 5
C
C      READ FIRST RECORD AND ALLOCATE CONTIGUOUS FILE
      READ(4,1001) IBLOK,FILNAM
1001  FORMAT(I3,10A1)
      CALL SETFIL(1, FILNAM, IERR, 'SY', 0)
      DEFINE FILE 1(IBLOK, 32, U, INDEX)
C
C      INITIALIZE ALL RECORDS OF OUTPUT FILE TO BLANKS
      DO 10 I=1,IBLOK
10    WRITE (1:I) NULL
C
C      READ INPUT AND WRITE EACH RECORD OF OUTPUT FILE
      READ (4,1002,END=50) I1,A
1002  FORMAT(I3,32A2)
      IF (I1 .LT. 0) GO TO 50
      I1=I1+1
      COUNT=COUNT+1
      WRITE (1:I1) A
      GOTO 40
C
C      READ THE CONTIGUOUS FILE AND
C      PRINT A LISTING OF THE COMPLETED FILE
50    CALL DATE(TODAY)
      WRITE (5,1003) FILNAM, TODAY
1003  FORMAT(29X,10A1/1H0,'MSG MESSAGE',17X,9A1/' NUM'/)
C
      DO 51 I1=1,IBLOK
      READ(1:I1) A
      I2=I1-1
51    WRITE (5,1004) I2,A
1004  FORMAT(X,I3,X,32A2)
      WRITE (5,1005) IBLOK,COUNT
1005  FORMAT(/X,I3,' MESSAGES ALLOCATED.',I4,' MESSAGES INPUT.')
```

The input file, ABC.SRC is:

```
010ABC.TST
002 FORT000002 SUBROUTINE DIRECTLY(INDIRECTLY) REFERENCES ITSELF
003 FORT000003 ILLEGAL FLOATING POINT INSTRUCTION
004 FORT000004 SYSTEM ERROR NO DIAGNOSTIC MESSAGE ASSIGNED
000 FORT000000 INVALID CALL TO ERROR
001 FORT000001 NO SPACE TO DO I/O
005 FORT000005 SYSTEM ERROR NO DIAGNOSTIC MESSAGE ASSIGNED
006 FORT001000 SYSTEM ERROR NO DIAGNOSTIC MESSAGE ASSIGNED
007 FORT001001 DEVICE PARITY ERROR
008 FORT001002 CHECKSUM/PARITY ERROR-END OF DATA ERROR (RANDOM)
010 FORT001004 END OF FILE OR END OF MEDIUM
009 FORT001003 UNDIAGNOSABLE I/O ERROR
010 FORT001004 END OF FILE OR END OF MEDIUM
012 FORT001006 DEFINE FILE NOT DONE (RANDOM ACCESS)
013 FORT001007 DEFINE FILE DONE (NOT RANDOM ACCESS)
015 FORT001009 FILE DOES NOT EXIST / OR IS ALREADY OPEN
016 FORT001010 UNABLE TO OPEN FILE
```

Before execution of the program, the disk contains:

DIRECTORY DF0: [200,200]

12-JUL-72

FORDGN.FTN	13	12-JUL-72	<233>
ABC .SRC	7	12-JUL-72	<233>
FORDGN.OBJ	30	12-JUL-72	<233>
FORDGN.LDA	86	12-JUL-72	<233>

TOTL BLKS:	136
TOTL FILES:	4

The program is run and the output file ABC.TST is created:

```
$ASSIGN SY:ABC.SRC,4
$RUN FORDGN.LDA
```

```
FORDGN V002A
INPUT FROM LOGICAL UNIT 4
$RUN PIP
PIP-11 V006A
```

```
#/DI
```

DIRECTORY DFB: [200,200]

12-JUL-72

```
FORDGN.FIN      13  12-JUL-72 <233>
ABC .SRC        7  12-JUL-72 <233>
FORDGN.OBJ      30  12-JUL-72 <233>
FORDGN.LDA      86  12-JUL-72 <233>
ABC .TST        9C  12-JUL-72 <233>
```

```
TOTL BLKS:    145
TOTL FILES:     5
```

The listing of the file produced on the line printer is:

ABC.TST

MSG MESSAGE
NUM

12-JUL-72

```
0  FORT000000 INVALID CALL TO ERROR
1  FORT000001 NO SPACE TO DO I/O
2  FORT000002 SUBROUTINE DIRECTLY(INDIRECTLY) REFERENCES ITSELF
3  FORT000003 ILLEGAL FLOATING POINT INSTRUCTION
4  FORT000004 SYSTEM ERROR NO DIAGNOSTIC MESSAGE ASSIGNED
5  FORT000005 SYSTEM ERROR NO DIAGNOSTIC MESSAGE ASSIGNED
6  FORT001000 SYSTEM ERROR NO DIAGNOSTIC MESSAGE ASSIGNED
7  FORT001001 DEVICE PARITY ERROR
8  FORT001002 CHECKSUM/PARITY ERROR-END OF DATA ERROR (RANDOM)
9  FORT001003 UNDIAGNOSABLE I/O ERROR
10 FORT001004 END OF FILE OR END OF MEDIUM
11
12 FORT001006 DEFINE FILE NOT DONE (RANDOM ACCESS)
13 FORT001007 DEFINE FILE DONE (NOT RANDOM ACCESS)
14
15 FORT001009 FILE DOES NOT EXIST / OR IS ALREADY OPEN
16 FORT001010 UNABLE TO OPEN FILE
17
```

18 MESSAGES ALLOCATED. 16 MESSAGES INPUT.

CHAPTER 11

FORTRAN OPERATING ENVIRONMENT

11.1 FORTRAN OBJECT TIME SYSTEM

The FORTRAN Object Time System (OTS) is composed of the following:

- a. Math routines, including all standard FORTRAN functions plus other arithmetic routines needed to do arithmetic operations (e.g., floating point);
- b. Miscellaneous utility routines (PDUMP, SETERR and SETFIL);
- c. I/O Routines, which handle the various types of FORTRAN I/O;
- d. Error handling routines, which process arithmetic errors, I/O errors and system errors; and
- e. Miscellaneous (Polish) routines required by the compiled code: \$SBS, \$DOEND, \$POP, etc.

The Library was designed as a large number of small pieces so that unnecessary routines could be omitted at link time. Thus, if the user performs only sequential formatted I/O, none of the random access routines are linked to his program.

11.2 THREADED CODE

Most FORTRAN compiled statements generate calls to Library sub-programs. These calls are based on the technique of evaluating expressions in Polish notation. Polish notation breaks down expressions into a large number of simple operations performed in a linear sequence. These operations use the PDP-11 stack for evaluating all expressions and subexpressions.

The implementation of Polish notation makes several assumptions:

- a. The first operation done in a Polish sequence is always a "push";
- b. It is not necessary to place breakpoints (as in ODT) in the middle of an arithmetic statement; and

- c. Speed does not suffer by assignment of a register (R4) for special purposes.

As an example, the FORTRAN statements:

```
A=1.0  
B=1.0
```

generate the following code:

```
;A=1.0  
$P00001  
.GLOBL $POP3  
$POP3,A  
;B=1.0  
$P00001  
$POP3,B
```

Routines such as those referenced, directly or indirectly, are inserted into the user program and are found at the end of the assembly listing; other routines are linked to the user program from the FORTRAN Library. The routines inserted into the user program as a result of the above code are as follows:

```
$P00001:  MOV    # $R00000+4,R0    ;GET ADDRESS OF VALUE  
          BR     $F00001  
  
$R00000:  04020000              ;FLOATING-POINT CONSTANT 1.0  
          00000000  
  
$F00001:  MOV    -(R0),-(SP)      ;PUSH 2-WORD VALUE ONTO  
          MOV    -(R0),-(SP)      ;STACK  
          JMP    @(R4)+           ;GO TO NEXT ROUTINE
```

The routine \$POP3 is in the FORTRAN Library. \$POP3 pops a real value off the stack into the memory location whose address follows the call to \$POP3 in the threaded code.

```
$POP3,A
```

pops the value on top of the stack into the two memory words reserved for A. Similarly,

```
$POP3,B
```

saves the two-word value found on top of the stack in B. To consider another example, the statement:

```
A = B+C
```

generates code similar to the following:

```

$P0001      ;PUSHES THE VARIABLE B ONTO THE STACK.
$P0002      ;EACH OPERATION CONSISTS OF THE ADDRESS OF A
.GLOBL $ADR  ;ROUTINE TO BE EXECUTED. A PUSH PLACES A
$ADR        ;VALUE ON THE STACK; A POP REMOVES A VALUE.
$POP3,A     ;THIS LINE CAUSES AN EXIT FROM POLISH MODE;
.+2         ;NORMAL EXECUTION IS RESUMED.

```

where \$P0001 and \$P0002 push the values of B and C onto the stack; \$ADR is a FORTRAN Library routine to add two floating-point numbers; \$POP3 saves the result in the variable A. The subroutine calls for the above sequence are as follows:

```

$P0001:  MOV  #B+4,R0      ;GET THE ADDRESS OF B.
          BR   $F0001      ;JUMP TO PUSH.
$P0002:  MOV  #C+4,R0      ;GET THE ADDRESS OF C.
$F0001:  MOV  -(R0),-(SP)   ;PUSH TWO
          MOV  -(R0),-(SP)  ;WORDS ON STACK.
          JMP  @(R4)+       ;JUMP TO NEXT ROUTINE.
$POP3:   MOV  (R4)+,R3      ;GET ADDRESS OF
                          ;VARIABLE DESTINATION.
          MOV  (SP)+,(R3)+  ;POP A VALUE
          MOV  (SP)+,(R3)+  ;FOR THE VARIABLE
          JMP  @(R4)+       ;GO TO NEXT ROUTINE.

```

Note that the instruction:

```
JMP @(R4)+
```

jumps to the next routine in the list as well as incrementing R4 over that item in the threaded code. All internal functions are called in this manner and exit using such a jump instruction and must clear any stack space used (except for the return value left on top of the stack).

All routines explicitly called by the user (i.e., subroutines and external functions) are called using the PDP-11 subroutine calling convention (see Section 6.4.7).

11.2.1 Entry to Polish Mode

Entry to Polish mode is made via a call to the \$POLSH routine which looks as follows:

```
JSR R4,$POLSH
```

which invokes the routine:

```
$POLSH; TST      (SP)+      ;DELETE OLD VALUE OF
                        ;R4 PUSHED ON ENTRY
                        JMP    @(R4)+ ;EXECUTE NEXT THREADED CODE CALL
```

The word following the JSR to \$POLSH is the first word of Polish code to be executed.

11.2.2 Exit from Polish Mode

An explicit exit from Polish mode is made by means of a word containing the address of the following word. For example:

```
.
.
.      ;IN POLISH MODE
.WORD  .+2      ;LEAVE POLISH MODE
.      ;CONTROL PASSES TO THIS LOCATION
.
.
```

Polish mode of execution is exited for execution of subroutine and function calls by means of the standard PDP-11 calling sequence (see Section 6.4.7). For example, if the statements:

```
A = B+C
CALL SUB(ARG)
```

were executed, the code following the

```
$POP3,A
```

(as shown at the end of Section 11.2) looks as follows:

```
.+2
JSR R5,SUB
BR  .+4
ARG
.
.
.
```

11.3 FORTRAN RUN-TIME MEMORY ORGANIZATION

The memory map during execution of a FORTRAN program is shown in Figure 11-1.

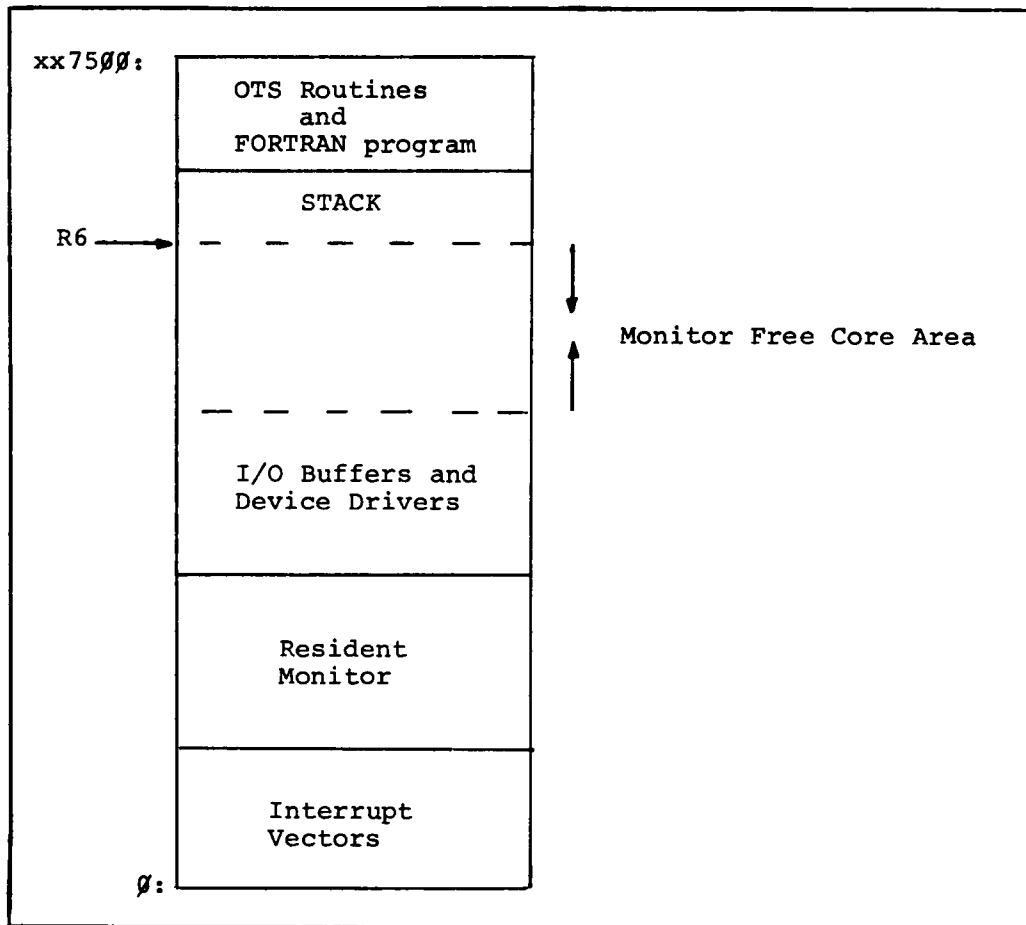


Figure 11-1

Memory Map Organization

As files are opened, space is taken from the Monitor free core area to accommodate additional I/O buffers and device drivers. The FORTRAN program and OTS routines use the processor R6 stack for storage of temporary results. Overflow of the stack into the areas dynamically allocated by the Monitor may cause DOS fatal errors or system crashes.

I/O buffer space and device drivers are released and the memory reclaimed when the logical unit is closed by an ENDFILE statement. ENDFILE may be used on both input and output files.

DOS I/O uses space from the free core area as follows:

1. One device driver routine for each device which has been initialized (.INIT) but not released (.RLSE). (The ENDFILE statement performs a .CLOSE and .RLSE). The approximate DOS driver sizes are given in Table 11-1.
2. One I/O buffer for each open dataset (FORTRAN logical unit in use). The buffer size depends upon the device.
3. One Device Data Block (DDB) for each open dataset (16 words).
4. One File Information Block (FIB) for each open file (16 words).
5. One bit map for each unit on which a file is open for output (64 words).

Table 11-1

Approximate DOS Device Driver Sizes

Device Specifier	Device Driver Size (words)	I/O Buffer Size (words)
DK	130	256
DF	70	64
DC	70	64
DT	190	256
MT	460	256
LP	160	48
CR (ASCII only)	210	48
KB	600	32
PP	80	32
PR	70	32

11.4 DEBUGGING FORTRAN PROGRAMS

The use of traditional FORTRAN debugging techniques such as PDUMP are recommended for development of FORTRAN applications (see Section 6.5.1).

System error reporting and traceback information is of significant value in debugging. Figure 11-2 is an example of the traceback feature in the FORTRAN system.

At run time, diagnostics are printed by OTS with a trace of the flow of control within the user-written code. Following the error code are printed the headings NAME and SEQ, below which are the names of the routines through which the call is being traced and the sequence number of the specific line in which the error occurred (or from which that subroutine was called). The first name and sequence number at the top of the list is the error location, subsequent names and numbers refer to the path through which the program reached that point.

The example in Figure 11-2 shows the same error message being printed twice. Though the two errors were generated by the same line in the subroutine, they can be traced to separate lines in the main program.

It requires considerable experience to successfully use ODT with a FORTRAN program. ODT was not intended to handle the problems of FORTRAN debugging, and it is especially hard to use when trying to debug threaded code.

The threaded code generated by the Compiler is a sequence of addresses, rather than machine instructions. ODT breakpoints can be placed only on machine instructions. The user is, therefore, constrained to use breakpoints only in places where the code leaves Polish Mode, e.g., a subroutine or function call. Alternatively, breakpoints can be placed in the Polish routines themselves.

There is one significant inconvenience associated with putting breakpoints in Polish routines. Polish routines are usually called from several places in a program and when the breakpoint in a Polish

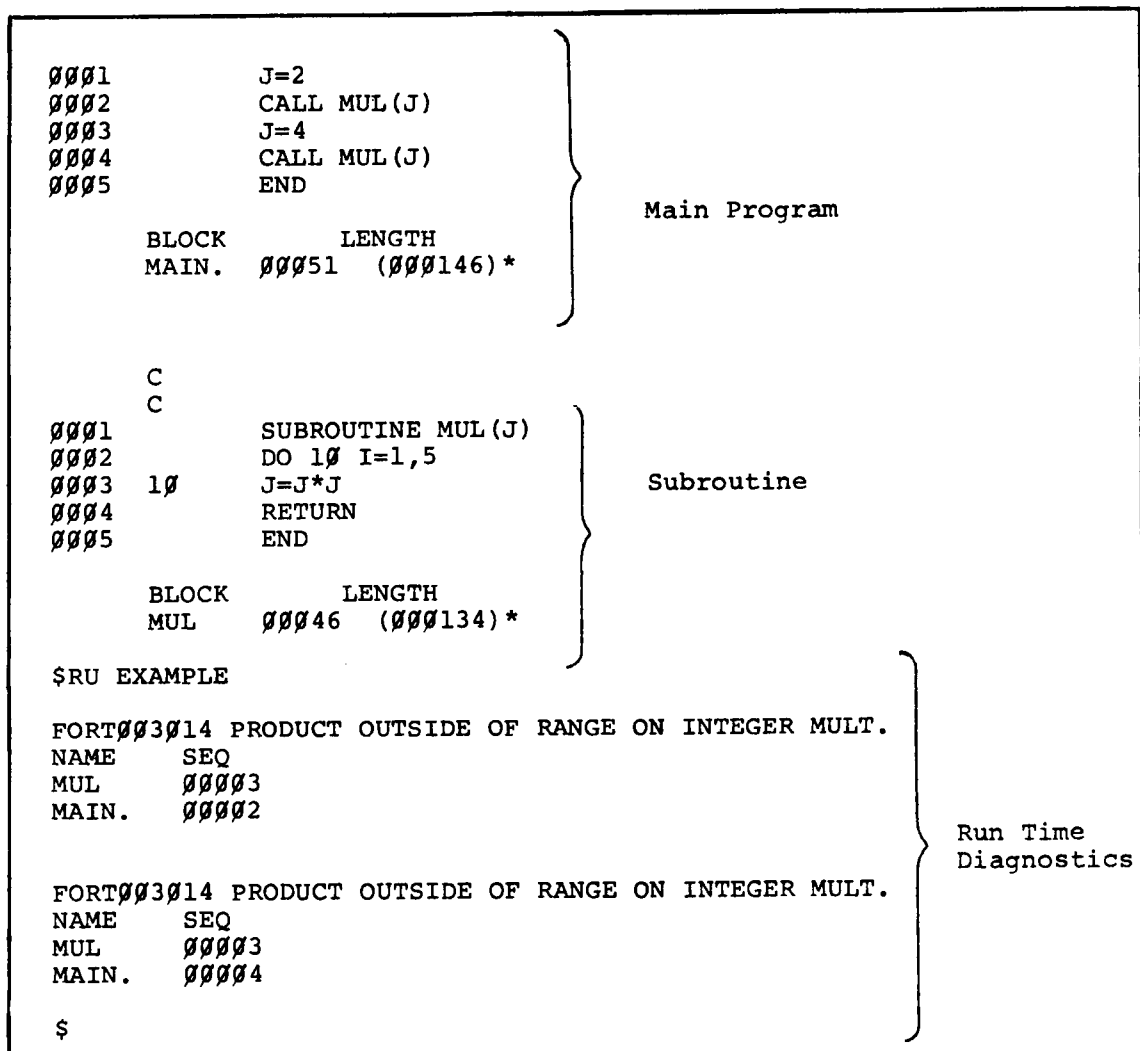


Figure 11-2
Example of Run Time Diagnostics

routine is encountered, the user must look at R4 to find the address from which the routine was called.

When executing a FORTRAN program which stores data outside the memory area allocated for an array, the Monitor messages F342 and F344 may occur and the system may halt. The Compiler /CK switch (Section 12.2) causes all subscript references to be checked for upper and lower bounds. Use of /CK is recommended while debugging FORTRAN programs.

11.5 FORTRAN OTS ERROR PROCESSING

The FORTRAN OTS error diagnostics are divided into 9 classes (Class 0-8). The messages associated with each diagnostic are given in Appendix E.

The FORTRAN OTS maintains an error class table which governs the actions taken for each error class. The user can modify the values in the error class table and hence the OTS error-processing by means of the SETERR subroutine call (Section 6.5.2).

Each error class entry contains values for the following variables:

COUNT	-	a count of the number of errors which have occurred for this class.
FLAG	-	set to 1 on any occurrence of an error in the class. May be tested and reset by means of CALL TSTERR (Section 6.5.9).
MAX	-	an integer specifying the action to occur on an error in the class.

The actions taken by the OTS when an error condition occurs are determined by the value of MAX as follows:

MAX>0	Set FLAG, log error message on output device, increment COUNT, and call EXIT if COUNT>MAX.
MAX=0	Set FLAG, log error message on output device, and continue program execution.
MAX=-1	Set FLAG, do not log message, ignore count, and continue program execution.
MAX=-2	Do not log message, EXIT to monitor, and print I351 message.

MAX=-3

Immediate exit to Monitor with a message of the form:

F030 0CC0NN

where CC and NN are the error class and number (in octal) of the condition causing the exit (see Appendix E).

The error classes and default MAX values are detailed in Table 11-2

Table 11-2
FORTRAN OTS Error Classes

Class	Default MAX Value	Meaning
0	-3	fatal errors; it is recommended that the occurrence count (-3) not be changed.
1	0	physical I/O errors.
2	1	errors in FORMAT statements.
3	3	arithmetic overflow or division by 0.
4	4	incorrect arguments to library functions or subroutines.
5	-1	arithmetic underflow errors.
6	0	conversion errors.
7	7	subscripting errors.
8	1	errors involving incorrect linkages to OTS routines; the maximum occurrence count (1) cannot be changed, although no diagnostic is given.

For example, arithmetic underflows (class 5) are ordinarily ignored by the FORTRAN OTS; arithmetic overflows (class 3) are logged and the default error limit is 3 occurrences.

CHAPTER 12

OPERATING PROCEDURES

12.1 USING THE FORTRAN SYSTEM

Figure 12-1 outlines the steps required to prepare a FORTRAN source program for execution under the DOS Monitor: (1) compilation, (2) linking, and (3) execution.

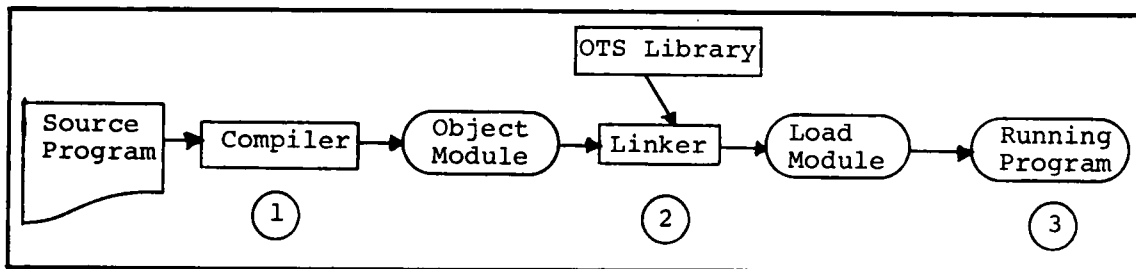


Figure 12-1

Steps in Compiling and Executing a FORTRAN Program

Step 1 in Figure 12-1 consists of a call to the FORTRAN Compiler to which the user passes a command string describing input and output files and possibly containing switch options. Step 2 consists of a call to the Linker to which the user passes a command string describing input and output files and possibly switch options. Steps 2 and 3 can be performed automatically following step 1 by including the /GO switch option in the FORTRAN command string. The /GO switch automatically causes the Linker to be called, the program linked, and execution begun.

12.1.1 Filename Specifications

The PDP-11 FORTRAN Compiler accepts a standard DOS command string, of the form:

output files < input file

The DOS command string syntax is described more completely in the DOS Monitor manual, Section 3.4.1.

The FORTRAN Compiler can produce two output files: an object module file and a listing file. The single input file permitted is a FORTRAN source file, containing one or more FORTRAN program units (a FORTRAN subprogram or main program). Each file specification is of the form:

dev:filename.ext[uic]

where dev: is any legal device specification code. The usual DOS device specifications are shown in Table 12-1.

Table 12-1
Device Specification Codes

dev:	Device
SY:	system disk, assumed default device
DKn:	RK11 disk, cartridge unit n; n=Ø is the default condition
DC:	RC11 disk
DF:	RF11 disk
KB:	user terminal
CR:	card reader
LP:	line printer
PR:	high-speed paper tape reader
PP:	high-speed paper tape punch
DTn:	DECTape unit n; n=Ø is the default condition
MTn:	Magtape unit n; n=Ø is the default condition.
BI:	BATCH input data set

The filename can be any one to six character alphanumeric name. The filename extension can be any one to three character alphanumeric sequence. The default assumed and supplied filename extensions for the FORTRAN Compiler (and for Link-11) are shown in Table 12-2.

Table 12-2

Filename Extensions

File	Assumed Extension on Input File	Default Extension on Output File
<u>FORTTRAN Compiler</u>		
object file	—	.OBJ
listing file	—	.LST
source file	.FTN	—
<u>Link-11</u>		
load module	—	.LDA
object module	.OBJ	—

Where a source file of the given filename (having no extension) cannot be found on the device specified using the default extension, the system attempts to find the filename with a null extension.

12.1.2 Compilation and Linking Procedures

To activate the FORTRAN Compiler under the DOS Monitor, the command:

```
$RUN FORTRN
```

is given. The FORTRAN Compiler then prints "FORTRAN", the Compiler version number and a # to indicate that it is ready to accept a command string. (All characters printed by the system are underlined to differentiate them from characters typed by the user.)

An example FORTRAN command sequence is shown below:

```
$RUN FORTRN
FORTRAN V004A
#OBJECT,LIST<FILE1
```

This command string directs the Compiler to take the source file FILE1.FTN from the system device and output the files LIST.LST and OBJECT.OBJ to the system device. The user then calls the PDP-11 Linker to link the object module(s) with the FORTRAN Library as follows:

(1) Link

```
$RUN LINK  
LINK-11 V011A  
#LOAD<OBJECT,FTNLIB/L/E
```

(2) Link and Execute

```
$RU LINK  
LINK-11 V011A  
#LOAD<OBJECT,FTNLIB/L/GO
```

Input to Link-11 is one or more compiled object modules and any required library files. Output from Link-11 is a single load module and, optionally, a load map and symbol table file.

In the linkage shown in case (1) above, the Linker output is a load module which can then be loaded and run by means of the RUN command. The /L switch indicates that FTNLIB is a library file and the /E switch indicates the end of input to the Linker. In case (2) shown above the /GO switch to the Linker causes the load module to be output to the system device and a copy of that load module loaded and execution begun as though a

\$RUN LOAD

command had been given. When the /GO switch is used, the /E switch is omitted from the Linker command string.

If any of the FORTRAN source files contains more than a single subroutine or a single main program, the resulting object module produced by the FORTRAN Compiler is a concatenated object module file. When such a file is linked by Link-11, the Linker /CC switch must be used following the object module file specification. Failure to use the /CC switch with a concatenated object module results in a W322 message from Link-11 and a defective load module.

The following sequence might be used to compile, link and execute a FORTRAN system consisting of:

- a. the FORTRAN main program MAIN.FTN,
- b. the FORTRAN subroutine SUBR1.FTN, and
- c. several FORTRAN subroutines in the file UTILITY.FTN.

```

$RUN FORTRN
FORTRAN V0004A
#MAIN,LP:<MAIN
#SUBR1,LP:<SUBR1
#UTILITY,LP:<UTILITY
#↑C
.KI
$RUN LINK
LINK-11 V011A
#P1,LP:<MAIN,SUBR1,UTILITY/CC,FTNLIB/L/E
#↑C
.KI
$RUN P1

```

Either of the Compiler output files can be omitted by omitting the file specification from the command string. For example:

```

$RUN FORTRN
FORTRAN V0004A
#FILE1<FILE1

```

produces FILE1.OBJ on the system device but no listing file.

```

# ,LP:<FILE1

```

produces a listing on the line printer, but no object module output.

12.1.3 Compile-Load-and-Go Operation

Rather than giving separate command strings to both the FORTRAN Compiler and to Link-11, a single command string to the FORTRAN compiler can be used to cause compilation, linking and execution. This one-step compile-load-and-go sequence is performed as follows:

```

$RUN FORTRN
FORTRAN V0004A
#FILE1,LIST<FILE1/GO

```

in which case the source program FILE1.FTN on the system device is compiled and linked to the FORTRAN Library (FTNLIB). An object file (FILE1.OBJ) and a load module file (FILE1.LDA) are created on the system device. The load module file is then loaded and execution begun.

12.1.4 FORTRAN Library Usage

The user can construct his own libraries of machine language and FORTRAN routines with the DOS Librarian, which he then searches at link time. The user is constrained to search all of his own libraries before searching the FORTRAN System Library.

Users should not add subroutines or functions to the DEC-supplied FORTRAN Library. Instead, users should create separate library files, using the Librarian. Thus, if MATLIB is a user library containing matrix manipulation routines, and the user writes a program (PROG) which uses routines from MATLIB, a command string to the Linker might be:

```
#PROG,LP:<PROG,MATLIB/L,FTNLIB/L/E
```

Routines should not be added to FTNLIB. Many routines in the OTS Library are position dependent and the insertion of new routines could result in undefined global references when linking FORTRAN programs. Similarly, the deletion or rearrangement of routines in FTNLIB is likely to cause similar problems unless the user is familiar with the ordering dependencies of the FORTRAN Library.

12.2 SWITCH OPTIONS

FORTRAN command strings can contain switch options on the input and output file specifications. Output switches are as follows and are initialized to the specified default values for each command string:

object/AS	The output produced by the Compiler is assembly code, rather than an object module. The default extension for the file is then .PAL rather than .OBJ.
list/LI:n	The listing control switch takes one of four arguments: n=0, minimal listing file: error diagnostics and block summaries only n=1, (default) source listing and block summaries n=2, source listing, assembly code, and block summaries n=3, source listing, assembly code, block summaries, and symbol table

The following switches are permitted, in any order and combination, on the input side of the FORTRAN command string:

/CK	Code is generated to check that all subscripts are within the bounds dimensioned in the program. Lower bounds for array subscripts (≥ 1) are always checked. Upper bound checking is optional and is caused by /CK.
/CO:n	A maximum of n continuation lines are permitted in the program. 99>n>0; the default value is n=5.
/ER	This switch causes the printing of special advisory diagnostics by the Compiler in the listing file. These S diagnostic codes flag questionable or some non-ANSI statements.
/GO	Causes a compile-load-and-go sequence. The FORTRAN Compiler automatically calls Link-11; Link-11 runs the program. Note that if F class errors occur during a compile, this switch is turned off and an EXECUTION DELETED message is printed at the console Teletype.
/ON	One word of storage is allocated for all INTEGER variables instead of two (the default case). This conserves space but may cause problems with EQUIVALENCE statements. All integers in PDP-11 FORTRAN have only one word of precision, even if two words are allocated for each value.
/SU	Suppresses generation of statement sequence numbers for use by the error traceback routine at run-time. This improves execution speed and saves two words of storage for each statement. Run-time error messages identify the subroutine in which an error occurred but not the line number at which it occurred.

All input switches, once set, remain set until a \$RESTART command is issued to the Monitor or another \$RUN FORTRN is performed.

As an example of the use of Compiler switch options:

```
$RUN FORTRN
FORTRAN V004A
#FILE1,LP:/LI:3<FILE1/CK/GO
```

performs a compile-load-and-go sequence, causes the Compiler to generate full run-time checking on array boundaries, and outputs a full source and assembly language listing on the line printer.

12.3 FORTRAN OUTPUT LISTING FORMAT

Where a listing file is requested, each page (when finally printed) contains a header with the following information:

Compiler version identification
current time
current date
listing page number

Using the /LI:n switch option, the user can vary the type of listing file obtained; from mere listing of error diagnostics through complete listing of source, assembly language and symbol table. All listing files include block summaries describing the current program unit.

The block summary or descriptor block is a collection of information about a FORTRAN main program or subroutine printed following the END statement on the source listing (before the assembly listing, if one is requested). Figures 12-2 and 12-3 show sample FORTRAN programs and their descriptor blocks. A descriptor block provides the following data:

- a. a listing of the switches specified in the command string.
- b. a listing of the routines called from the program.
- c. a listing of the size of the program in both decimal words and octal bytes. Sizes are also provided for each common block declared within the program. Since common blocks may be shared among several programs, their sizes are of individual interest.

The information is formatted into three columns under two headings: BLOCK and LENGTH. The name of the program or common block is listed under BLOCK, followed by the size in decimal words, followed by the size in octal bytes in parentheses. These last two entries are under the LENGTH heading.

The line containing the entry with the program name is followed by an asterisk. Where no program name is supplied, the name MAIN. is shown. The name .\$\$\$\$. denotes the presence of blank common; the name DATA. indicates the presence of a block data subprogram.

- d. Following the Assembly listing, where one is requested, the final data item supplied, entitled:

****COMPILER ----- CORE****

has three subheadings: PHASE, USED, and FREE.

```

0001      SUBROUTINE SUB2X(I)
0002      DOUBLE PRECISION D05
0003      COMMON I1, Z, D05 /L2/ J2(10), V2, W2
0004      I1=1
0005      Z=-1.0
0006      D05=-1.001
0007      W2 = SQRT(EXP(J2(I)-1.0))
0008      V2 = -1.0
0009      RETURN
0010      END

```

ROUTINES CALLED:

SQRT , EXP

SWITCHES = /ON,/CK

BLOCK	LENGTH
SUB2X	34 (000274)*
\$.SSSS.	7 (000016)
L2	24 (000050)

```

**COMPILER ----- CORE**
      PHASE      USED  FREE
DECLARATIVES 00355 13275
EXECUTABLES  00543 13099
ASSEMBLY     00955 20594

```

Figure 12-2

Block Summary Example #1

```

0001      BLOCK DATA
0002      COMMON /L1/ XBD, YBD, CBD, DBD /L2/ UBD, VBD, WBD
0003      COMPLEX CBD
0004      DOUBLE PRECISION DBD
0005      DATA XBD, YBD, CBD, DBD, UBD, VBD, WBD / 2*1.0,(1.0,1.0),1.0D+2,
0006      1 3*1.0 /
0007      END

```

BLOCK	LENGTH
DATA.	7 (000000)*
L1	12 (000030)
L2	5 (000014)

```

**COMPILER ----- CORE**
      PHASE      USED  FREE
DECLARATIVES 00355 13275
EXECUTABLES  00551 13092
ASSEMBLY     00845 20714

```

Figure 12-3

Block Summary Example #2

These terms head columns containing, respectively, the phase of the Compiler (three phases), the number of decimal words of core storage used in that phase, and the number of decimal words of core storage free in that phase.

These sizes describe the symbol table space used in each phase and provide the user with a ratio of used to unused space in the Compiler.

If the FORTRAN program was compiled without switches or subroutine calls, the information described in (a) or (b), respectively, is not provided. The information in (c) and (d) is provided as part of the listing for every FORTRAN program or subroutine.

12.4 COMPILE-TIME MEMORY REQUIREMENTS

The PDP-11 FORTRAN Compiler uses all available memory space for the program symbol table. A simple variable entry in the symbol table is eight words long. A constant entry is eight words plus the size of the constant. An array entry is ten words plus one word for each dimension. For example, a complex constant entry is twelve words long and a 3-dimensional array entry is thirteen words long. The user can reduce the amount of memory used by the Compiler by his choice of input/output files in command strings and by reducing the number of continuation lines permitted. Memory space saved in this way is available as additional symbol table space.

The /CO switch, which controls the number of allowable continuation lines on a given FORTRAN statement, takes space from the symbol table area. Thus, /CO:99 takes the maximum amount of space from the symbol table while /CO:Ø makes additional space available to the symbol table. (The default value is /CO:5).

The devices used for I/O affect the Compiler core requirements. The minimum core required for I/O operations would be used with the command sequence:

```
$RUN FORTRN  
FORTRAN VØØ4A  
#FILE<FILE
```

where source and object files are on the system device. Memory requirements are increased and symbol table space correspondingly reduced when the user requests listings and uses devices other than the system device. For example:

```
#FILE,KB:<DT:FILE
```

requires considerably more memory space than the previous command string. This is a problem of some significance for smaller FORTRAN systems.



APPENDIX A
FORTRAN CHARACTER SET

A.1 ASCII CHARACTER SET

EVEN PARITY BIT	7-BIT OCTAL CODE	CHARACTER	REMARKS
0	000	NUL	NULL, TAPE FEED, CONTROL/SHIFT/P.
1	001	SOH	START OF HEADING: ALSO SOM, START OF MESSAGE, CONTROL/A.
1	002	STX	START OF TEXT; ALSO EOA, END OF ADDRESS, CONTROL/B.
0	003	ETX	END OF TEXT; ALSO EOM, END OF MESSAGE, CONTROL/C.
1	004	EOT	END OF TRANSMISSION (END); SHUTS OFF TWX MACHINES, CONTROL/D.
0	005	ENQ	ENQUIRY (ENQRY); ALSO WRU, CONTROL/E.
0	006	ACK	ACKNOWLEDGE; ALSO RU, CONTROL/F.
1	007	BEL	RINGS THE BELL. CONTROL/G.
1	010	BS	BACKSPACE; ALSO FEO, FORMAT EFFECTOR. BACKSPACES SOME MACHINES, CONTROL/H.
0	011	HT	HORIZONTAL TAB. CONTROL/I.
0	012	LF	LINE FEED OR LINE SPACE (NEW LINE); ADVANCES PAPER TO NEXT LINE, DUPLICATED BY CONTROL/J.
1	013	VT	VERTICAL TAB (VTAB). CONTROL/K.
0	014	FF	FORM FEED TO TOP OF NEXT PAGE (PAGE). CONTROL/L.
1	015	CR	CARRIAGE RETURN TO BEGINNING OF LINE. DUPLICATED BY CONTROL/M.
1	016	SO	SHIFT OUT; CHANGES RIBBON COLOR TO RED. CONTROL/N.
0	017	SI	SHIFT IN; CHANGES RIBBON COLOR TO BLACK. CONTROL/O.
1	020	DLE	DATA LINK ESCAPE. CONTROL/P (DC0).
0	021	DC1	DEVICE CONTROL 1, TURNS TRANSMITTER (READER) ON, CONTROL/Q (X ON).
0	022	DC2	DEVICE CONTROL 2, TURNS PUNCH OR AUX ON. CONTROL/R (TAPE, AUX ON).
1	023	DC3	DEVICE CONTROL 3, TURNS TRANSMITTER (READER) OFF, CONTROL/S (X OFF).
0	024	DC4	DEVICE CONTROL 4, TURNS PUNCH OR AUX OFF. CONTROL/T (AUX OFF).
1	025	NAK	NEGATIVE ACKNOWLEDGE; ALSO ERR, ERROR. CONTROL/U.
1	026	SYN	SYNCHRONOUS FILE (SYNC). CONTROL/V.
0	027	ETB	END OF TRANSMISSION BLOCK; ALSO LEM, LOGICAL END OF MEDIUM. CONTROL/W.
0	030	CAN	CANCEL (CANCL). CONTROL/X.
1	031	EM	END OF MEDIUM. CONTROL/Y.
1	032	SUB	SUBSTITUTE. CONTROL/Z.
0	033	ESC	ESCAPE. PREFIX. CONTROL/SHIFT/K.
1	034	FS	FILE SEPARATOR. CONTROL/SHIFT/L.
0	035	GS	GROUP SEPARATOR. CONTROL/SHIFT/M.
0	036	RS	RECORD SEPARATOR. CONTROL/SHIFT/N.
1	037	US	UNIT SEPARATOR. CONTROL/SHIFT/O.
1	040	SP	SPACE.
0	041	!	
0	042	"	
1	043	#	
0	044	\$	
1	045	%	
1	046	&	
0	047	'	APOSTROPHE.
0	050	(	
1	051	)	
1	052	*	
0	053	+	
1	054	,	
0	055	.	

EVEN PARITY BIT	7-BIT OCTAL CODE	CHARACTER	REMARKS	EVEN PARITY BIT	7-BIT OCTAL CODE	CHARACTER	REMARKS
0	056	.		1	130	X	
1	057	/		0	131	Y	
0	060	0		0	132	Z	
1	061	1		1	133	[	SHIFT/K.
1	062	2		0	134	\	SHIFT/L.
0	063	3		1	135	]	SHIFT/M.
1	064	4		1	136	↑	*
0	065	5		0	137	←	**
0	066	6		0	140	`	ACCENT, GRAVE
1	067	7		1	141	a	
1	070	8		1	142	b	
0	071	9		0	143	c	
0	072	:		1	144	d	
1	073	;		0	145	e	
0	074	<		0	146	f	
1	075	=		1	147	g	
1	076	>		1	150	h	
0	077	?		0	151	i	
1	100	@		0	152	j	
0	101	A		1	153	k	
0	102	B		0	154	l	
1	103	C		1	155	m	
0	104	D		1	156	n	
1	105	E		0	157	o	
1	106	F		1	160	p	
0	107	G		0	161	q	
0	110	H		0	162	r	
1	111	I		1	163	s	
1	112	J		0	164	t	
0	113	K		1	165	u	
1	114	L		1	166	v	
0	115	M		0	167	w	
0	116	N		0	170	x	
1	117	O		1	171	y	
0	120	P		1	172	z	
1	121	Q		0	173	{	
1	122	R		1	174		
0	123	S		0	175	}	THIS CODE GENERATED BY ALTMODE
1	124	T					THIS CODE GENERATED BY ESC KEY (IF PRESENT)
0	125	U		0	176	~	DELETE, RUB OUT
0	126	V					
1	127	W		1	177	DEL	

*↑ appears as ^ on some machines.

**← appears as _ (underscore) on some machines.

A.2 PDP-11 PUNCHED CARD CODES

CHARACTER	Parity ASCII	DECØ29	DECØ26	CHARACTER	Parity ASCII	DECØ29	DECØ26
{	173	12 Ø	12 Ø	@	3ØØ	8 4	8 4
}	175	11 Ø	11 Ø	A	1Ø1	12 1	12 1
SPACE	24Ø	NONE	NONE	B	1Ø2	12 2	12 2
!	Ø41	12 8 7	12 8 7	C	3Ø3	12 3	12 3
"	Ø42	8 7	Ø 8 5	D	1Ø4	12 4	12 4
#	243	8 3	Ø 8 6	E	3Ø5	12 5	12 5
\$	Ø44	11 8 3	11 8 3	F	3Ø6	12 6	12 6
%	245	Ø 8 4	Ø 8 7	G	1Ø7	12 7	12 7
&	246	12	11 8 7	H	11Ø	12 8	12 8
'	Ø47	8 5	8 6	I	311	12 9	12 9
(	Ø5Ø	12 8 5	Ø 8 4	J	312	11 1	11 1
)	251	11 8 5	12 8 4	K	113	11 2	11 2
*	252	11 8 4	11 8 4	L	314	11 3	11 3
+	Ø53	12 8 6	12	M	115	11 4	11 4
,	254	Ø 8 3	Ø 8 3	N	116	11 5	11 5
-	Ø55	11	11	O	317	11 6	11 6
.	Ø56	12 8 3	12 8 3	P	12Ø	11 7	11 7
/	257	Ø 1	Ø 1	Q	321	11 8	11 8
Ø	Ø6Ø	Ø	Ø	R	322	11 9	11 9
1	261	1	1	S	123	Ø 2	Ø 2
2	262	2	2	T	324	Ø 3	Ø 3
3	Ø63	3	3	U	125	Ø 4	Ø 4
4	264	4	4	V	126	Ø 5	Ø 5
5	Ø65	5	5	W	327	Ø 6	Ø 6
6	Ø66	6	6	X	33Ø	Ø 7	Ø 7
7	267	7	7	Y	131	Ø 8	Ø 8
8	27Ø	8	8	Z	132	Ø 9	Ø 9
9	Ø71	9	9	[	333	12 8 2	11 8 5
:	Ø72	8 2	11 8 2	\	134	Ø 8 2	8 7
;	273	11 8 6	Ø 8 2	]	335	11 8 2	12 8 5
<	Ø74	12 8 4	12 8 6	† or ^	336	11 8 7	8 5
=	275	8 6	8 3	+ or _	137	Ø 8 5	8 2
>	276	Ø 8 6	11 8 6				
?	Ø77	Ø 8 7	12 8 2				



1

2



3

4

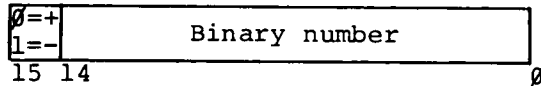


APPENDIX B

FORTTRAN WORD FORMATS

B.1 INTEGER FORMAT

Sign



Integers are stored in a two's complement representation. In 2-word format, an integer is assigned two storage words, although only the high-order word (i.e., the word having the lower address) is significant. Use of the /ON switch (see Section 12.2) causes integers to be assigned a single storage word. Integer constants must lie in the range -32768 to +32767. For example:

$$\begin{aligned} +22 &= 000026_8 \\ -7 &= 177771_8 \end{aligned}$$

B.2 FLOATING-POINT FORMATS

The exponent for both 2-word and 4-word floating-point formats is stored in excess 128 (200_8) notation. Exponents from -128 to +127 are represented by the binary equivalents of 0 through 255 (0 through 377_8). Fractions are represented in sign magnitude notation with the binary radix point to the left. Numbers are assumed to be normalized and, therefore, the most significant bit is not stored because it is redundant (this is called "hidden bit normalization"). This bit is assumed to be a 1 unless the exponent is 0 (corresponding to 2^{-128}) in which case it is assumed to be 0. The value 0 is represented by two or four words of zeroes. For example, $+1.0$ would be represented by:

40200
0

in the 2-word format, or:

40200
0
0
0

in the 4-word format. -5 would be:

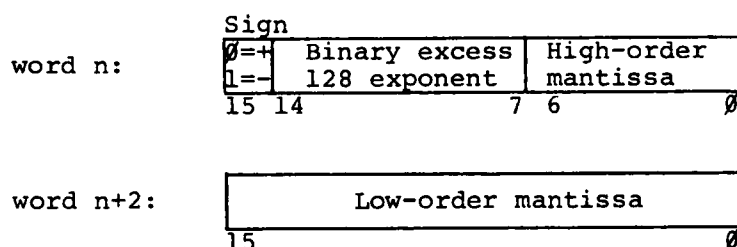
140640
0

in the 2-word format, or:

140640
0
0
0

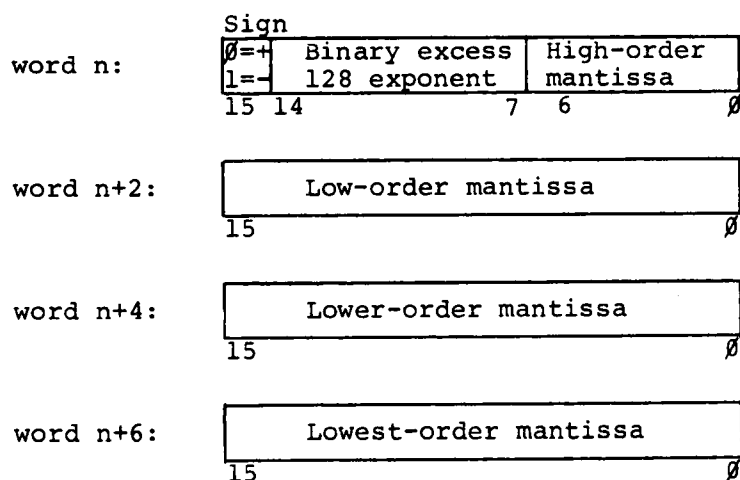
in the 4-word format.

B.2.1 Real Format (2-Word Floating Point)



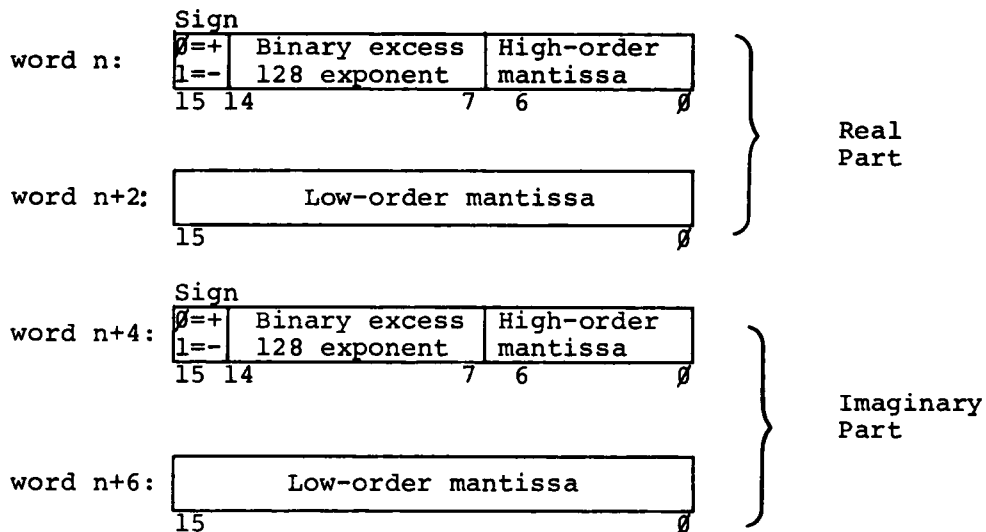
Floating-point numbers are stored in a sign magnitude representation. This format is limited to normalized numbers. Since the high-order bit of the mantissa is always 1, it is discarded, giving an effective precision of 24 bits, or approximately 7 digits of accuracy. The magnitude range lies between approximately 0.14×10^{-38} and 1.7×10^{38} .

B.2.2 Double Precision Format (4-Word Floating Point)

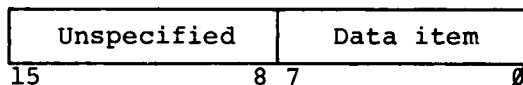


The effective precision is 56 bits or approximately 16 decimal digits of accuracy. The magnitude range lies between 0.14×10^{-38} and 1.7×10^{38} .

B.3 COMPLEX FORMAT

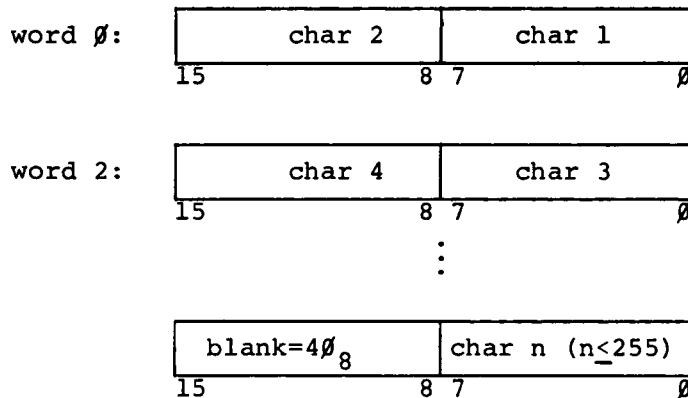


B.4 BYTE FORMAT



The range of numbers from +127 to -128 can be represented in BYTE (LOGICAL *1) format. BYTE format array elements are stored in adjacent bytes.

B.5 HOLLERITH FORMAT



Hollerith constants are stored internally one character per byte, filling up to word boundaries. An odd number of characters causes a blank to be appended to the constant. A Hollerith constant included in an arithmetic expression is treated as a one-word integer quantity where only the first two characters of the constant are significant.

B.6 LOGICAL FORMAT

True:

1	7	7	7	7	7
---	---	---	---	---	---

15Ø

False:

Ø	Ø	Ø	Ø	Ø	Ø
---	---	---	---	---	---

15Ø

Logical format data items are treated as one-word integer values for use with arithmetic and logical operators. Any non-zero value is considered to have a logical value of true when tested by a logical IF statement.

APPENDIX C

FORTRAN LANGUAGE SUMMARY

C.1 OPERATOR SUMMARY

Operators in each type are shown in order of descending priority.

<u>Type</u>	<u>Operator</u>	<u>Operates Upon</u>
Arithmetic	**	exponentiation
	-	numeric constants
	*, /	variables and ex- pressions
	+, -	
	=	
Relational		
	.GT.	greater than
	.GE.	greater than or equal to
	.LT.	less than
	.LE.	less than or equal to
Logical	.EQ.	equal to
	.NE.	not equal to
	.NOT.	.NOT.A is true if and only if A is false and is false if A is true.
	.AND.	A.AND.B is true if and only if A and B are both true and is false if either A or B is false.
	.OR.	A.OR.B is true if and only if either A or B or both are true and false if both A and B are false.

C.2 SYSTEM SUBROUTINE SUMMARY

<u>System Subroutine</u>	<u>Function</u>	<u>Manual Section</u>
DATE	returns the current system date.	6.5.6
EXIT	causes a return to the Monitor.	6.5.5
LINK	requests the loading of an overlay file.	7.5
PDUMP	causes specified portions of core to be dumped onto the error logging device.	6.5.1
RAN RANDU	returns a random number.	6.5.4
RETURN	transfers control to the statement following the last CALL LINK executed from the resident section.	7.5
RUN	loads and initiates execution of another program.	7.12
SETERR	allows the user to reset the maximum occurrence count for any class of run-time error.	6.5.2
SETFIL	overrides the default value for logical device assignment at run time	6.5.3
SSWTCH	tests specified bits of the console switch register.	6.5.8
TIME	returns the current system time.	6.5.7
TSTERR	allows the user program to monitor the types of errors detected during program execution.	6.5.9

C.3 FORTRAN FUNCTION SUMMARY

Form	Definition	Argument Type	Result Type
ABS(X)	Real absolute value	Real	Real
IABS(I)	Integer absolute value	Integer	Integer
DABS(X)	Double precision absolute value	Double	Double
CABS(Z)	Complex to Real, absolute value where $Z = (x, y)$ $CABS(Z) = (x^2 + y^2)^{1/2}$	Complex	Real
FLOAT(I)	Integer to Real conversion	Integer	Real
IFIX(X)	Real to Integer conversion IFIX(X) is equivalent to INT(X)	Real	Integer
SNGL(X)	Double to Real conversion	Double	Real
DBLE(X)	Real to Double conversion	Real	Double
REAL(Z)	Complex to Real conversion obtain real part	Complex	Real
AIMAG(Z)	Complex to Real conversion, obtain imaginary part	Complex	Real
CMPLX(X,Y)	Real to Complex conversion $CMPLX(X,Y) = X + iY$	Real	Complex
Truncation functions return the sign of the argument * largest integer $\leq arg $			
AINTE(X)	Real to Real truncation	Real	Real
INT(X)	Real to Integer truncation	Real	Integer
IDINT(X)	Double to Integer truncation	Double	Integer
Remainder functions return the remainder when the first argument is divided by the second.			
AMOD(X,Y)	Real remainder	Real	Real
MOD(I,J)	Integer remainder	Integer	Integer
DMOD(X,Y)	Double precision remainder	Double	Double
Maximum value functions return the largest value from among the argument list; ≥ 2 arguments.			
AMAX0(I,J,...)	Real maximum from Integer list	Integer	Real
AMAX1(X,Y,...)	Real maximum from Real list	Real	Real
MAX0(I,J,...)	Integer maximum from Integer list	Integer	Integer
MAX1(X,Y,...)	Integer maximum from Real list	Real	Integer
DMAX1(X,Y,...)	Double maximum from Double list	Double	Double
Minimum value functions return the smallest value from among the argument list; ≥ 2 arguments.			
AMIN0(I,J,...)	Real minimum of Integer list	Integer	Real
AMIN1(X,Y,...)	Real minimum of Real list	Real	Real
MIN0(I,J,...)	Integer minimum of Integer list	Integer	Integer
MIN1(X,Y,...)	Integer minimum of Real list	Real	Integer
DMIN1(X,Y,...)	Double minimum of Double list	Double	Double
The transfer of sign functions return (sign of the first argument) * (absolute value of second argument).			
SIGN(X,Y)	Real transfer of sign	Real	Real
ISIGN(I,J)	Integer transfer of sign	Integer	Integer
DSIGN(X,Y)	Double precision transfer of sign	Double	Double

Form	Definition	Argument Type	Result Type
	Positive difference functions return the first argument minus the minimum of the two arguments.		
DIM(X,Y)	Real positive difference	Real	Real
IDIM(I,J)	Integer positive difference	Integer	Integer
	Exponential functions return the value of e raised to the argument power.		
EXP(X)	e^X	Real	Real
DEXP(X)	e^X	Double	Double
CEXP(Z)	e^Z	Complex	Complex
ALOG(X)	Returns $\log_e(X)$	Real	Real
ALOG10(X)	Returns $\log_{10}(X)$	Real	Real
DLOG(X)	Returns $\log_e(X)$	Double	Double
DLOG10(X)	Returns $\log_{10}(X)$	Double	Double
CLOG(Z)	Returns $\log_e$ of complex argument	Complex	Complex
SQRT(X)	Square root of Real argument	Real	Real
DSQRT(X)	Square root of Double precision argument	Double	Double
CSQRT(Z)	Square root of Complex argument	Complex	Complex
SIN(X)	Real sine	Real	Real
DSIN(X)	Double precision sine	Double	Double
CSIN(Z)	Complex sine	Complex	Complex
COS(X)	Real cosine	Real	Real
DCOS(X)	Double precision cosine	Double	Double
CCOS(Z)	Complex cosine	Complex	Complex
TANH(X)	Hyperbolic tangent	Real	Real
ATAN(X)	Real arc tangent	Real	Real
DATAN(X)	Double precision arc tangent	Double	Double
ATAN2(X,Y)	Real arc tangent of (X/Y)	Real	Real
DATAN2(X,Y)	Double precision arc tangent of (X/Y)	Double	Double
CONJG(Z)	Complex conjugate, if $Z=X+i*Y$ $CONJG(Z)=X-i+Y$	Complex	Complex
RAN(I,J)	Returns a random number of uniform distribution over the range 0 to 1. I and J must be integer variables and should be set initially to 0. Resetting I and J to 0 regenerates the random number sequence. Alternate starting values for I and J will generate different random number sequences. See also Section 6.5.4.	Integer	Real

Sine and cosine functions take arguments expressed in radians.

C.4 SUMMARY OF FORTRAN STATEMENTS

The following summary of statements available in the PDP-11 FORTRAN IV language defines the general format for the statement. If more detailed information is needed, the reader is referred to the Section(s) in the manual dealing with that particular statement.

<u>Statement Formats</u>	<u>Effect</u>	<u>Manual Section</u>
<u>Arithmetic Assignment</u>		
variable = expression		3.1
	the value of the arithmetic or logical expression is assigned to the variable.	3.2
<u>Arithmetic Statement Functions</u>		
name(var ₁ ,var ₂ ,...) = expression		6.3
	creates a user-defined function having the variables var _i as dummy arguments. When referenced, the expression is evaluated using the actual arguments in the function call.	
<u>ASSIGN</u>		
ASSIGN n TO ivar		3.3
	set the integer variable, ivar, equal to the statement number, n.	
<u>BACKSPACE</u>		
BACKSPACE u		9.6.1
	logical unit number u is backspaced one record.	
<u>BLOCK DATA</u>		
BLOCK DATA		6.4.8
	specifies the subprogram which follows as a BLOCK DATA subprogram.	

<u>Statement Formats</u>	<u>Effect</u>	<u>Manual Section</u>
<u>CALL</u>		
CALL name		6.4.5
CALL name (arg ₁ ,arg ₂ ,...)		
	Call the SUBROUTINE subprogram with the name specified; passing the actual arguments, arg _i , to replace the dummy arguments in the SUBROUTINE definition.	
<u>COMMON</u>		
COMMON /name1/var ₁ ,var ₂ ,.../name2/var _j ,var _k		5.4
		5.5.2
	reserves one or more blocks of storage space under the name specified to contain the variables var _i associated with that block name.	7.8.4
		7.12
<u>CONTINUE</u>		
CONTINUE		4.4
	causes no processing.	
<u>DATA</u>		
DATA var ₁ ,var ₂ ,.../val ₁ ,val ₂ ,.../		5.7
	causes elements in the list of values, val _i , to be initially stored in the corresponding elements of the list of variable names, var _i .	
<u>DECODE</u>		
DECODE (c,f,v)list		9.7
	change the elements in the list of variables from ASCII into the desired internal format; c specifies the number characters, f specifies the format, and v is the name of an array containing the ASCII characters to be converted.	

<u>Statement Formats</u>	<u>Effect</u>	<u>Manual Section</u>
<u>DEFINE FILE</u>		
DEFINE FILE u(m,n,U,ivar)		9.4.1
	defines the record structure of a disk or DEctape file where u is the logical unit number, m is the number of records in the file, n is the length in words of a single record, U is a fixed argument, and ivar is the associated variable pointing to the next record.	
<u>DIMENSION</u>		
DIMENSION s ₁ ,s ₂ ,...,s _k		5.3 6.4.2
	reserves storage space for the arrays specified by s _i where s _i is the array variable name followed by one, two, or three integer constants in parentheses.	
<u>DO</u>		
DO n ivar = m ₁ ,m ₂ ,m ₃		4.3
	1. Set ivar=m ₁ 2. Execute statements through n 3. Increment ivar=ivar+m ₂ 4. If ivar≤m ₃ , go to 2, above	
<u>END</u>		
END		4.7
	delimits a program unit.	
<u>ENCODE</u>		
ENCODE (c,f,v)list		9.7
	change the elements in the list of variables into ASCII format; c specifies the number of characters in the buffer, f specifies the format, and v is the name of the resulting array to be used as a buffer.	
<u>END FILE</u>		
END FILE u		9.6.3
	the file currently open on logical unit number u is closed.	

<u>Statement Formats</u>	<u>Effect</u>	<u>Manual Section</u>
<u>EQUIVALENCE</u>		
EQUIVALENCE (var ₁ ,var ₂ ,...), (var _j ,var _k ,...)		5.5
	each of the variables within a set	5.5.1
	of parentheses is assigned the same	5.5.2
	storage location.	5.5.3
<u>EXTERNAL</u>		
EXTERNAL name1,name2,...		5.6
	informs the system that the names	
	specified are those of FUNCTION or	
	SUBROUTINE subprograms.	
<u>FIND</u>		
FIND (u'r)		9.4.2
	included for compatibility with	
	other FORTRAN systems. This state-	
	ment is ignored by the system.	
<u>FORMAT</u>		
n FORMAT(field specification,...)		8.1
	describes the format in which one	
	or more records are to be transmitted;	
	a statement number, n, must be specified.	
<u>FUNCTION</u>		
FUNCTION name (var ₁ ,var ₂ ,...)		6.4.3
type FUNCTION name (var ₁ ,var ₂ ,...)		
	begins a FUNCTION subprogram, indicating	
	the program name and any dummy variable	
	names, var _i . An optional type specifi-	
	cation can be included.	
<u>GOTO</u>		
GOTO n		4.1.1
	transfer control to statement number n.	

<u>Statement Formats</u>	<u>Effect</u>	<u>Manual Section</u>
<u>GOTO</u>		
Computed		
GOTO (n ₁ ,n ₂ ,...,n _k), ivar		4.1.2
	transfer control to the statement number, n _i , where i=ivar.	
Assigned		
GOTO ivar		4.1.3
GOTO ivar, (n ₁ ,n ₂ ,...,n _k)		
	transfer control to the statement number currently associated with ivar.	
<u>IF</u>		
Arithmetic		
IF (expression) n ₁ ,n ₂ ,n ₃		4.2.1
	transfer control to statement number n _i depending upon the value of the expression.	
Logical		
IF (expression) statement		4.2.2
	execute the statement if the logical expression tests true.	
<u>IMPLICIT</u>		
IMPLICIT type a ₁ , (b ₁ -b ₂),...		5.1
	the elements a _i and b _i represent single (or a range of) letter(s) whose presence as the initial letter of a variable specifies the variable to be of that type.	
<u>PAUSE</u>		
PAUSE		4.5
PAUSE octal		
	suspend program execution and print the octal constant if one is specified.	

<u>Statement Formats</u>	<u>Effect</u>	<u>Manual Section</u>
<u>READ</u>		
Formatted		9.2.1
READ (u,f) READ (u,f) list READ f,list	causes a read operation on device u according to format specification f and assigning values to the variables in the list.	
Unformatted		9.3.1
READ (u) READ (u) list	causes a read operation on the device u, assigning values to the variables in the list.	
Direct Access		9.4.3
READ (u'r) list	causes a read operation on logical unit number u of record number r, assigning values to the variables in the list.	
Transfer of Control on Error		9.5
END=n ERR=n END=n,ERR=n	is an optional element in each type of READ statement allowing the program to transfer to statement number n.	
<u>RETURN</u>		
RETURN	returns control to the calling program from the current subprogram.	6.4.8
<u>REWIND</u>		
REWIND u	logical unit number u is repositioned to the beginning of the currently opened file.	9.6.2

<u>Statement Formats</u>	<u>Effect</u>	<u>Manual Section</u>
<u>STOP</u>		
STOP		4.6
STOP octal		
	terminate program execution and print the octal constant if one is specified.	
<u>SUBROUTINE</u>		
SUBROUTINE name		6.4.4
SUBROUTINE name (var ₁ ,var ₂ ,...)		
	begins a SUBROUTINE subprogram, indicating the program name and any dummy variable names, var _i .	
<u>Type Declaration</u>		
type var ₁ ,var ₂ ,...,var _k		5.2
	the variable names, var _i , are assigned the specified data type ⁱ in the program unit.	
<u>WRITE</u>		
Formatted		9.2.2
		9.2.3
WRITE (u,f)		
WRITE (u,f) list		
PRINT f,list		
	causes one or more logical records containing the values of the variables in the list to be written onto device u, according to the format specification f.	
Unformatted		9.3.2
WRITE (u)		
WRITE (u)list		
	causes one or more logical records containing the values of the variables in the list to be written onto device u.	

<u>Statement Formats</u>	<u>Effect</u>	<u>Manual Section</u>
<u>WRITE</u>		
Direct Access		9.4.4
WRITE (u'r) list	causes one or more logical records containing the values of the variables in the list to be written onto record r of logical unit number u.	
Transfer of Control on Error		9.5
END=n ERR=n END=n,ERR=n	is an optional parenthetical element in each type of WRITE statement allowing the program to transfer to statement number n on an end-of-file or error condition.	

APPENDIX D

FORTRAN LIBRARY FUNCTIONS

This Appendix contains a brief outline of the OTS library of FORTRAN functions which involve approximations. "Floating point" means single precision, 2-word, floating point format with a 24-bit fraction and an 8-bit binary exponent. "Double precision" means 4-word, floating point format with a 56-bit fraction and an 8-bit binary exponent. The values of the coefficients used in the various approximations may be found at the cited parts of the following references:

- a. Computer Approximations, by J. F. Hart et al, John Wiley & Sons, 1968.
- b. Approximation for Digital Computers, by C. Hastings et al, Princeton University Press, 1955.
- c. PDP-11 Paper Tape Software Programming Handbook, DEC-11-GGPC-D, Digital Equipment Corporation.

In the descriptions of the various functions, the relative error values given are for the approximating polynomials in the cited intervals. These error bounds assume exact arithmetic. There are two additional sources of errors in the function calculations which are not considered in the error bounds:

- a. Rounding and truncation errors can occur in reducing the given argument to the range in which the polynomial or rational fraction approximations are valid.
- b. Rounding errors can occur as a result of using finite precision floating-point arithmetic in the polynomial or rational fraction computations.

All OTS FORTRAN functions are called using the standard sequence:

```
JSR      R5,NAME
BR       RTN
.WORD    #ARG1,...,#ARGn
RTN:
```

The result is returned in R0-R1 for floating point and R0-R3 for a double precision function.

Some FORTRAN functions call other single argument functions in the course of their computation. In order for them to be reentrant, these calls are made via the routine \$FCALL.

```

MOV    ARGUMENT ADDRESS, R5
MOV    #FUNCTION NAME, R4
JSR    PC,$FCALL

```

\$FCALL calls the FORTRAN function whose address is in R4 with the argument whose address is in R5. Control is returned to the instruction following the JSR with the function result in R0-R1 for floating point or R0-R3 for double precision.

D.1 ALOG(X), Floating Point Natural Logarithm

If $X \leq 0$ call error

Let $X = y \cdot 2^a$ where $1/2 \leq y < 1$

Let $Q = (y \sqrt{2} - 1) / (y \sqrt{2} + 1)$

Then $\ln(X) = a \cdot \ln(2) + \ln(y)$

$$\text{ALOG}(X) = a \cdot \ln(2) = \ln(\sqrt{2}) + \sum_{i=0}^3 c_i Q^{2i}$$

where the c_i are drawn from Hart #2662. The relative error is $\leq 10^{-9.9}$.

D.2 ALOG10(X), Floating Point Common Logarithm

Computed as $\log_{10}(e) \cdot \text{ALOG}(X)$.

D.3 ATAN(X), Floating Point Arctangent

If $X < 0$, $\text{ATAN}(X) = -\text{ATAN}(-X)$

If $|X| > 1$, $\text{ATAN}(|X|) = \pi/2 - \text{ATAN}(1/|X|)$

If $|X| > \tan \pi/12$, $\text{ATAN}(X) = \pi/6 + \text{ATAN}((X\sqrt{3} - 1)/(X + \sqrt{3}))$

For $|X| \leq \tan \pi/12$, $\text{ATAN}(X) = \sum_{i=0}^4 c_i X^{2i}$

where the c_i are drawn from Hart #4941. The relative error is $\leq 10^{-9.5}$

D.4 ATAN2(X,Y), Two Argument Floating Point Arctangent

If $Y = 0$, or $X/Y > 2^{25}$, $ATAN2(X,Y) = \pi/2$ (sign X).
If $Y > 0$, and $X/Y \leq 2^{25}$, $ATAN2(X,Y) = ATAN(X/Y)$
If $Y < 0$, and $X/Y \leq 2^{25}$, $ATAN2(X,Y) = \pi \cdot \text{sign} X + ATAN(X/Y)$

D.5 DATAN(X), Double Precision Arctangent

The analysis is the same as in that for ATAN(X) except that the polynomial approximant is of degree 8. The coefficients are drawn from Hart #4945. The relative error is $\leq 10^{-16.8}$.

D.6 DATAN2(X,Y), Two Argument Double Precision Arctangent

The rules for DATAN2 are the same as those for ATAN2 except that the DATAN is used in all computations.

D.7 DLOG(X), Double Precision Natural Logarithm

The analysis for DLOG is the same as that for ALOG except that the polynomial in Q^2 is of degree 6. The c_i are drawn from Hart #2665. The relative error is $\leq 10^{-16.5}$.

D.8 DLOG10(X), Double Precision Common Logarithm

Computed as $\log_{10}(e) \cdot DLOG(X)$.

D.9 DSQRT(X), Double Precision Square Root

If $X \leq 0$ call error

Let $X = A \cdot 2^B$ where $1/2 \leq A < 1$

Let $Y_0 = 2^{B/2} \cdot (1/2 + A/2)$ if B is even

or

$Y_0 = 2^{(B+1)/2} \cdot (1/4 + A/2)$ if B is odd,

a transformation requiring only two instructions. Starting with Y_0 , four Newton-Raphson iterations are performed.

$$Y_{n+1} = 1/2(Y_n + x/Y_n).$$

The relative error is $< 10^{-17}$.

D.10 DSIN(X), Double Precision Sine

Let $y = \text{Integer}(4 \cdot \text{fraction}(X/2\pi))$
 Let $V = \text{Fraction}(4 \cdot \text{fraction}(X/2\pi))$
 Then $\text{DSIN}(X) = P(V\pi/2)$ if $y=0$
 $= P((1-V)\pi/2)$ if $y=1$
 $= P(-V\pi/2)$ if $y=2$
 $= P((V-1)\pi/2)$ if $y=3$

where $\sin(V\pi/2) \approx P(V\pi/2) = V \sum_{i=0}^8 c_i V^{2i}$ for $-1 \leq V \leq 1$

The c_i are drawn from Hart #3345. The relative error is $\leq 10^{-18.6}$

D.11 DCOS(X), Double Precision Cosine

Computed as $\text{DSIN}(X + \pi/2)$.

D.12 DEXP(X), Double Precision Exponential

If $X > 87$ call overflow
 If $|X| < 2^{-60}$, $\text{DEXP}(X) = 1$
 If $X < -88.7$, $\text{DEXP}(X) = 0$
 Let $y = \text{Integer}(X \cdot \log_2(e))$
 Let $V = 16 \cdot \text{Fraction}(X \cdot \log_2(e))$
 Let $w = 1/16 \cdot \text{Fraction}(V)$
 $\text{DEXP} = 2^y \cdot 2^w \cdot 2^{\text{Integer}(V)/16}$ where $0 \leq w < 1/16$.

Powers of $2^{1/16}$ are obtained from a table.

$$2^w \approx \frac{P(w^2) + wQ(w^2)}{P(w^2) - wQ(w^2)} \quad \text{where } P \text{ and } Q \text{ are first}$$

degree polynomials in w^2 .

The coefficients of P and Q are drawn from Hart #1121.

The relative error is $\leq 10^{-16.4}$.

D.13 EXP(X), Floating Point Exponential

If $X > 87$, call overflow
 If $|X| < 2^{-28}$, $\text{EXP}(X) = 1$.
 If $X < -88.7$, $\text{EXP}(X) = 0$.
 Let $y = \text{Integer}(X \cdot \log_2(e))$
 Let $V = \text{Fraction}(X \cdot \log_2(e))$
 Let $w = 1/2 \ln(2) \cdot V$ where $|w| \leq \ln(2)/2$

Then $\text{EXP}(X) = 2^y \cdot (e^w)^2$

where $e^w \approx 1 + \frac{2 \cdot w}{\frac{c_1 - w - c_2}{c_3 + w^2}}$

The c_i are drawn from DEC-11-GGPC-D, page 7-23. The relative error is $\leq 10^{-10}$.

D.14 RAN(I1,I2) and CALL RANDU(I1,I2,F), Random Number Generator

RAN and RANDU use the same multiplicative congruential algorithm for generating uniformly distributed pseudo-random numbers in the interval (0,1).

If $I1=0$, $I2=0$ set generator base
 $X_{n+1} = 2^{16} + 3$
 otherwise
 $X_{n+1} = (2^{16} + 3) \cdot X_n \bmod 2^{32}$

Store generator base X_{n+1} in $I1, I2$.

Result is X_{n+1} scaled to a real value Y_{n+1} , $0 < Y_{n+1} < 1$.

D.15 SIN(X), Floating Point Sine

The analysis is the same as that for DSIN(X). The polynomial approximant used is of degree 4 and the coefficients are drawn from Hastings, sheet 16. The relative error is $\leq 2 \cdot 10^{-8}$.

D.16 COS(X), Floating Point Cosine

Computed as $\text{SIN}(X + \pi/2)$.

D.17 SQRT(X), Floating Point Square Root

The analysis is the same as that for DSQRT(X) except that only three iterations are performed. The relative error is $\leq 10^{-8}$.

D.18 TANH(X), Floating Point Hyperbolic Tangent

If $|X| \geq 16$, $\text{TANH}(X) \approx 1 \cdot \text{sign}(X)$
otherwise

let $y = \text{EXP}(2 \cdot X)$

$\text{TANH}(X) = (y-1)/(y+1)$.

APPENDIX E

FORTRAN ERROR DIAGNOSTICS

E.1 FORTRAN COMPILER ERROR DIAGNOSTICS

There are three basic Compiler error classes. They are:

- F - Fatal errors which must be corrected before the program can run correctly.
- W - Warning errors which should be corrected. The program is not likely to run with W class errors.
- I - Informational errors which consist of minor syntax errors which do not affect the code generated by the Compiler.

There is one other error class which is of interest to the user. This is the "S" class (special) message. Errors in this class are never issued unless the user specifies the /ER switch in the command string to the Compiler. S class errors are those errors which may be conflicts with ANSI usage or which may be errors only under unusual usages in the program. Each S error should be carefully considered within the context of the program. For most programs, these errors may be safely ignored.

Error diagnostics issued by the Compiler consist of two or three lines. The first line consists of a short section of the source line where the error was assumed to occur. The second line consists of the error number; and the third, if present, consists of the text of the error message. This text message is not printed if the Compiler error diagnostic file, FORCOM.DGN, is not present on the system disk under the [1,1] area.

The following is an example of a section of a FORTRAN program error diagnostic as issued by the Compiler:

```
00001      DIMENSION NLIST(1000)
00002      DIMENSION LIST (1000)
00003      DIMENSION A(5,5)
00004      DEFINE FILE LIST (100,10,U,IND)
```

```
[LE LIST (1)
ERROR 122
F ILLEGAL FORM FOR UNIT NUMBER IN DEFINE FILE.
```

Ø I REDUNDANT CONTINUATION MARK: IT IS IGNORED.

A continuation mark has been seen which does not make sense. The Compiler ignores it and continues as if the mark had not existed.

1 I CONTINUATION MARK NOT IN RANGE 1 TO 9; IT IS IGNORED.

This error occurs when a special character has been typed after a TAB in the free form input specifications. If a continuation is to be specified when using this form of Compiler input, the only meaningful form is numeric. If it is desired to use non-numeric characters for continuation characters, the standard column 6 convention should be used.

2 I ILLEGAL STMT. NUMBER, NON-NUMERIC CHAR. IN COLS. 1-5.

This error occurs when any unrecognizable forms occur in the statement number field of a line.

3 W ILLEGAL TYPE OR IMPLICIT STATEMENT, INTEGER IS ASSUMED.

This error is caused by the recognition of the optional word TYPE or the word IMPLICIT not followed by a recognizable type descriptor. The recognizable type descriptors are BYTE, LOGICAL*1, LOGICAL, LOGICAL*2, INTEGER, INTEGER*2, REAL, REAL*4, DOUBLE, DOUBLE PRECISION, REAL*8 and COMPLEX.

4 W NON-DECLARATIVE STATEMENT IN BLOCK DATA.

A statement other than a COMMON, EQUIVALENCE, DIMENSION, TYPE or DATA occurred within a BLOCK DATA subprogram.

5 F SYMBOL TABLE FULL.

The Compiler has run out of space to process the program. It is either necessary to reduce the number of variables, constants or arrays; cut down on the number of continuation lines requested; specify I/O devices which use less memory; or reduce the number of I/O devices in use.

6 W TOO MANY CONTINUATION LINES, REMAINDER IGNORED.

More continuation lines were used than were allowed in the compile. The default number of continuation lines is 5. If more or fewer are needed, specify the /CO:n switch in the command string where n is the number of continuations desired; n may range from 0 to 99. Note that the space assigned to continuation lines in the Compiler subtracts from the symbol table space. Each continuation line requested uses approximately the same space as 5 symbols in the symbol table.

7 S MIXED MODE IS USED IN AN EXPRESSION.

This error is only issued if the /ER switch has been specified. This error message flags all expressions where an implied mode conversion occurs.

8 F ILLEGAL UNARY OP., ONLY +, -, OR .NOT. ARE ALLOWED.

A binary operator was used in an expression as though it were a unary operator.

9 F CLOSING "/" MISSING ON BLOCK NAME.

The block name specifier in a COMMON statement is missing the final /. It is impossible to determine where the block name ended. This error is also issued if a block name is longer than 6 characters.

10 W ALL PORTIONS OF FORMAT MUST BE WITHIN OUTER LEVEL PARENS.

The only part of a FORMAT statement which can appear outside the enclosing parentheses is the key-word FORMAT.

11 W TOO MANY RIGHT PARENTHESES IN FORMAT.

There are mismatched parentheses in the FORMAT statement.

12 W ILLEGAL CHARACTER(S) TERMINATING A STATEMENT.

This error is issued whenever characters remain after the Compiler finishes processing a statement. This usually occurs with other errors, but may occur in expressions which have illegal characters or operations specified.

13 F ILLEGAL FORM FOR A NUMERIC CONSTANT.

The number being processed by the Compiler has an illegal form. For example, A "." is flagged as a bad constant.

14 F INSUFFICIENT COMPILER SPACE TO EVALUATE THIS CONSTANT.

The Compiler ran out of space while trying to evaluate a constant. This problem has the same remedy as error 5 (symbol table overflow).

15 W INTEGER CONSTANT TOO BIG. LARGEST POS. VALUE ASSUMED.

The number in question was not in the range -32768 to +32767. It has been assigned the value 32767.

16 F ILLEGAL SYNTAX IN LIST ITEM.

One of the declarative list items is improperly constructed; perhaps a subscript was omitted or an extra comma was added.

17 F ILLEGAL LIST TERMINATION.

The last declarative list item was not terminated by a carriage return/line feed. This may be caused by a variable name longer than 6 characters or an unrecognizable construct.

18 W ILLEGAL CHARACTER IN FORMAT STATEMENT.

A character was found in a FORMAT statement which could not be part of any legal format specification.

19 F DIMENSION MUST BE SPECIFIED FOR EACH VAR. IN THE LIST.

A variable was specified in a DIMENSION statement without the required dimension list.

20 F DIMS. CONFLICT WITH THOSE OF AN EARLIER STATEMENT.

A DIMENSION was declared which contradicts an earlier statement.

21 W UNRECOGNIZED STATEMENT.

The statement could not be recognized as a legal FORTRAN construct. It is probably misspelled. The Compiler made no attempt to compile it and has ignored it.

22 F ADJUSTABLE ARRAY NAME OR INDEX NOT A SUBPROGRAM PARAMETER.

Any adjustable array names or indices must be specified in the parameter list of the SUBROUTINE or FUNCTION statement. They are not allowed as local variables. They may not be specified in common.

23 W MISSING EXPONENT IN CONSTANT.

An E or D format constant was specified, but the exponent was omitted.

24 F MAXIMUM FUNCTION DEPTH (20) EXCEEDED.

In an expression, function calls may be nested to a maximum depth of 20.

25 F MISMATCHED PARENTHESES.

Mismatched parentheses were found in an expression.

26 F NON-ARRAY REFERENCE TO ARRAY ITEM.

This error occurs in an arithmetic expression where an item which has been declared as an array was referenced as a simple variable. All array element references must specify a distinct subscript expression.

27 F CANNOT ASSIGN TO A CONSTANT.

A constant appeared as the left side of an arithmetic expression.

28 F CANNOT ASSIGN TO A FUNCTION

A function call appeared as the left side of an arithmetic expression.

29 W ILLEGAL CHAR. TERMINATING A STMT. OR POSS. BAD OPERATOR.

Similar to error 12; characters remain after processing an expression. The probable cause is a bad operator in the expression.

30 F SUBSCRIPT ON NON-ARRAY VARIABLE.

Attempt to use a variable which had not been declared in a DIMENSION statement as an indexed variable or array.

31 W NAME MUST BE 1-6 ALPHANUMERICS, THE FIRST ALPHABETIC.

The name specified was longer than 6 characters or did not start with an alphabetic character.

32 F ILL. SUBSCRIPT IN AN ARRAY ASSIGNMENT OR MISSING SUBSCRIPT.

The subscript form used with an array variable was not recognized by the Compiler.

33 F ILLEGAL OPERAND OR POSSIBLE ADJACENT OPERATORS.

The character(s) following a legal operator in an expression could not be recognized as a legal operand in the expression. Two operators were probably placed together.

34 F TOO MANY SUBSCRIPTS OR NO CLOSING PAREN FOR SUBSCRIPT.

More than three subscripts were specified for a subscripted variable, or the closing parenthesis of the subscript was omitted.

35 F NO FUNCTION ARGUMENTS PRESENT.

All function calls must have at least one argument specified.

36 F UNRECOGNIZABLE PARAMETER IN FUNCTION CALL.

An argument in a function call could not be recognized.

37 F FUNCTION CALL MISSING A ")".

The closing parenthesis of a function call was missing. A possible reason was that an ASCII string without a closing quote mark was specified as one of the arguments.

38 F ILLEGAL ROUTINE NAME.

A function or subroutine name must consist of from 1 to 6 alphanumeric characters, the first of which must be alphabetic.

39 W MISSING END STATEMENT, END IS ASSUMED.

The Compiler did not find an END statement at the end of the file. The Compiler has inserted one to allow completion of the compile.

40 W IMPROPERLY NESTED DO STATEMENT.

A DO loop, when nested, cannot overlap the range of any other DO statement.

41 W DO LIST OVERFLOW, NO MORE THAN 10 NESTED DO'S ARE ALLOWED.

The Compiler table containing information about DO loop nesting has overflowed because an attempt was made to nest deeper than 10 DO loops.

42 W ILLEGAL SYNTAX IN COMMON/EQUIVALENCE.

A list item in a COMMON or EQUIVALENCE statement was incorrectly written.

43 W TABLE OVERFLOW IN COMMON/EQUIVALENCE.

The Compiler ran out of space while attempting to compile the statement. This error has the same solutions as error 5. Also, if possible, reduce the number of COMMON or EQUIVALENCE declarations used.

44 W DUMMY VARIABLE OR ADJUSTABLE ARRAY USED IN COMMON.

It is illegal to use a dummy variable in a COMMON declaration because of possible data allocation conflicts. Also, if adjustable arrays are specified, they cannot occur in common.

45 W VARIABLE ALREADY IN COMMON, CANNOT BE RE-DEFINED.

An attempt was made to place a variable in common which was already there.

46 F ILLEGAL DO STATEMENT SYNTAX.

The DO statement specified had one or more unrecognizable parts.

47 F DO CONTROL VARIABLE IS NOT SIMPLE INTEGER VARIABLE OR CONSTANT.

The control variable must be only an integer variable or constant; it cannot be an array or any type other than integer.

48 F DO PARAMETER IS NOT A SIMPLE INTEGER VARIABLE OR CONSTANT.

The initial or terminal value of a DO loop is not a simple integer variable or constant.

49 W BAD STEP VALUE IN DO, IT IS ASSUMED TO BE 1.

The step value of a DO loop is not a simple integer variable or constant. A value of 1 has been substituted.

50 W ILLEGAL CONSTANT IN PAUSE/STOP.

The constant in a PAUSE or STOP statement does not consist of a one to six digit octal constant with a value less than 177777.

51 W ILLEGAL OR MISSING STATEMENT LABEL, IT MUST BE NUMERIC.

The statement label field did not have a proper label consisting of 1 to 5 numeric digits.

52 F ILLEGAL SYNTAX IN GOTO/ASSIGN STATEMENT.

One or more unrecognizable items were found in a GOTO or ASSIGN statement.

53 W ILLEGAL PARAMETER IN STATEMENT FUNCTION LIST.

A parameter in an arithmetic statement function may consist only of a simple variable; no subscripted variables or constants are allowed.

54 F ROUTINE NAME CANNOT BE A NUMERIC CONSTANT.

The name of a subroutine or function must consist of 1 to 6 alphanumeric characters, the first alphabetic.

55 W "SUBROUTINE" OR "FUNCTION" NOT FIRST STMT. OF ROUTINE.

A SUBROUTINE or FUNCTION statement may not occur except as the first statement of a routine.

56 W ILLEGAL PARAMETER IN SUBROUTINE OR FUNCTION LIST.

A parameter in a subroutine or function list may consist only of simple variables; no subscripted variables or constants are allowed.

57 W TOO MANY PARAMETERS (>255) IN ROUTINE LIST.

A parameter list for a subroutine or function may have no more than 255 parameters specified.

58 W CONSTANT MAY NOT BE DECLARED IN EXTERNAL.

The EXTERNAL statement does not allow constants to be specified in the list.

59 W TOO MANY LEFT PARENTHESES IN FORMAT.

The parentheses in a FORMAT statement are not correctly matched.

60 W MISSING COMMA OR) IN COMMON/EQUIVALENCE.

An item described in a COMMON or EQUIVALENCE statement is terminated incorrectly.

61 W MISSING (IN COMMON/EQUIVALENCE.

A left parenthesis has been omitted in a COMMON or EQUIVALENCE statement.

62 W DUMMY ARGUMENT OR ADJUSTABLE ARRAY USED IN EQUIVALENCE.

It is illegal to attempt to equivalence an item to any item which was passed as a parameter to a subroutine or function.

63 W INCONSISTENT EQUIVALENCE.

An equivalence was specified which, if attempted, would produce inconsistent allocation of the variables in question.

64 F TWO OR MORE COMMON ITEMS ARE EQUIVALENCED.

Any item in common may be equivalenced only to an item not in common.

65 F I/O UNIT IS NOT SIMPLE INTEGER VARIABLE OR CONSTANT.

The I/O unit specified must be a simple integer variable or constant; it may not be an array element or be of any type other than integer.

66 F ARRAY OR FUNCTION NAME NOT ALLOWED AS UNIT IN I/O STMT.

The I/O unit specified must be a simple integer or constant; it may not be an array or function name or be of any type other than integer.

67 F ILLEGAL SYNTAX IN I/O OR ENCODE/DECODE.

An unrecognizable form was found in an I/O or ENCODE/DECODE statement.

68 F MISSING ARGUMENT IN FIND.

The FIND statement must have exactly two arguments.

69 F ILLEGAL RECORD DESIGNATOR IN RANDOM ACCESS READ/WRITE.

The record designator in a random access READ or WRITE must be a simple integer variable or constant.

70 F MISSING RIGHT PARENTHESIS IN I/O OR ENCODE/DECODE.

The closing parenthesis in the I/O, ENCODE or DECODE statement was omitted; or an unrecognizable parameter follows the last necessary parameter.

71 W ILLEGAL FORM FOR END= AND/OR ERR= .

The END= or ERR= specifications must refer to a legal 1 to 5 digit statement number.

72 W ILLEGAL FORM FOR LIST ITEM.

An I/O list item is unrecognizable. List items may consist of variables, arrays, subscripted variables and/or implied DO lists.

73 F ILLEGAL SYNTAX FOR REWIND, BACKSPACE, OR ENDFILE.

The REWIND, BACKSPACE and ENDFILE statements must specify a simple integer variable or constant as the logical unit number.

74 F NON-INTEGER PARAMETER IN REWIND, BACKSPACE, OR ENDFILE.

The parameter must be a simple integer variable or constant.

75 W ILLEGAL H CONSTANT IN FORMAT.

The Hollerith constant specified has been incorrectly formed.

76 W HOLLERITH CONSTANT COUNT TOO BIG.

The count in a Hollerith field cannot exceed 255.

77 W SYNTAX ERROR IN IMPLICIT STATEMENT.

The variable specification in the IMPLICIT statement has been incorrectly formed.

78 W HOLLERITH CONSTANT IMPROPERLY TERMINATED BY END OF LINE.

The Hollerith constant count was larger than the available number of characters on the line.

79 W .NOT. MAY BE USED AS A UNARY OPERATOR ONLY.

An attempt was made to use .NOT. as a binary operator.

80 W EXPONENT MAY NOT BE LOGICAL*1, LOGICAL*2 OR COMPLEX.

An attempt was made to do exponentiation with an exponent which was illegal.

81 W INTEGER**REAL OR COMPLEX**DOUBLE NOT ALLOWED.

The attempted form of exponentiation was illegal.

82 W COMPLEX**REAL OR COMPLEX**DOUBLE NOT ALLOWED.

The attempted form of exponentiation was illegal.

83 F IMPROPER LABEL SYNTAX IN IF STATEMENT.

A transfer label was specified in an IF statement which was not a 1 to 5 digit number.

84 W ANYTHING **COMPLEX NOT ALLOWED.

Complex exponentiation of this form is not allowed.

85 F MISSING COMMA IN READ, PRINT, OR PUNCH.

A comma must immediately follow the format number in the READ or PRINT statement.

86 F INCORRECT SYNTAX IN DEFINE FILE STATEMENT.

A DEFINE FILE was stated incorrectly.

87 W ARRAY MAY NOT BE DECLARED AS EXTERNAL.

Only functions or subroutines may be declared as EXTERNAL.

88 F ARRAY IS TOO LARGE.

An array was specified which could not possibly fit on a PDP-11.

89 F ILLEGAL ROUTINE NAME.

A routine name must consist of 1 to 6 alphanumeric characters, the first of which is alphabetic.

90 F ILLEGAL DO SPECIFICATION IN I/O OR ENCODE/DECODE.

The implied DO specification in the statement was incorrect.

91 F ILLEGAL LIST IN IMPLIED DO.

The I/O list inside the implied DO loop was incorrectly formed.

92 F ILLEGAL FORMAT SPEC. IN I/O OR ENCODE/DECODE.

An array specifier did not consist of a legal numeric format number or array name.

93 W SYNTAX ERROR IN THE EXPRESSION OF AN ASF.

The expression in an arithmetic statement function follows the same rules as a normal arithmetic statement.

94 W MISSING ",", OR ")" IN ASF.

A parameter was not followed by a comma or left parenthesis. The parameter may possibly be badly formed.

95 W MISPLACED "=" IN ASF.

The assignment portion of the arithmetic statement function is either missing or incorrectly specified.

96 F ASF NAME HAS BEEN PREVIOUSLY USED.

An arithmetic statement function must have a uniquely defined name.

97 W SUBSCRIPTS OUT OF BOUNDS IN DATA OR EQUIVALENCE.

An attempt was made to use DATA or EQUIVALENCE outside the range of an established array.

98 F ILLEGAL EXTENSION OF COMMON ORIGIN BY EQUIVALENCE.

The beginning of common cannot be extended using an EQUIVALENCE statement.

99 F OPENING "/" MISSING FROM DATA GROUP.

The start of the data group portion of a DATA statement could not be found.

100 W UNEQUAL NUMBER OF VARIABLES AND CONSTANTS.

When using the DATA statement, the number of constants specified must exactly match the number of variable or array items to be filled.

101 W DATA NOT ALLOWED IN COMMON EXCEPT IN "BLOCKDATA".

Presetting common cannot be done except in a BLOCK DATA subprogram.

102 F SUBSCRIPTS ON UNDIMENSIONED ELEMENT IN DATA.

A non-subscripted variable cannot be referenced with a subscript in any form.

103 F ADJUSTABLE ARRAY NOT ALLOWED IN DATA.

It is not legal to attempt to preset items passed as parameters to a subprogram.

104 F PRESETTING NAMED COMMON ALLOWED ONLY IN "BLOCKDATA".

A common block may be preset with a DATA statement only within a BLOCK DATA subprogram.

105 F ILLEGAL FORM FOR CONSTANT IN DATA.

The constant specified was not recognized by the data processor.

106 F ILLEGAL REPEAT COUNT.

The repeat count in a DATA statement was incorrectly specified.

107 W MISMATCHED DATA TYPES.

The data types specified do not match the variable types in a DATA statement.

108 W DATA MUST FOLLOW ALL OTHER DECLARATIVES.

The DATA statement must come after DIMENSION, COMMON, EQUIVALENCE or type declaration statements.

109 I NO PATH TO THIS STATEMENT.

It is not possible for the program to reach this statement. This is usually caused by an unlabelled statement immediately following a GOTO or IF form.

110 W VARIABLE MAY NOT HAVE BEEN REDEFINED AFTER USE IN "ASSIGN".

A variable was used in an expression which was previously used in an ASSIGN statement and has not been redefined.

111 F ILLEGAL FORM FOR COMPLEX CONSTANT.

A complex constant may consist only of a (, followed by a real constant, followed by a comma, followed by another real constant, followed by a).

112 W NUMBER OF DIMENSIONS NOT THE SAME AS DECLARED.

An attempt was made to use a subscripted variable with a different number of subscripts than was declared.

113 W RETURN IS ILLEGAL IN MAIN PROGRAM.

A RETURN statement is only legal within a subroutine or function.

114 W MISSING DO LOOP TERMINATION(S).

One or more DO loops were left unterminated in the program.

115 F DIMENS. NOT TERM. BY RIGHT PAREN OR TOO MANY DIMENSIONS.

A dimensioned variable may have, at most, three dimensions and must be terminated by a right parenthesis.

116 W ROUTINE NAME CANNOT BE DECLARED EXTERNAL.

A program or subroutine cannot be external to itself.

117 F ENCODE/DECODE LACKS LEGAL BUFFER DESCRIPTOR.

The buffer descriptor in an ENCODE or DECODE statement may consist only of the name of a legal array.

118 F ENCODE/DECODE BUFFER SIZE NOT SIMPLE INTEGER.

The buffer size descriptor in an ENCODE or DECODE statement must be a simple integer variable or constant.

119 F ARRAY OR FUNCTION CANNOT DESCRIBE BUFFER SIZE.

The buffer size descriptor in an ENCODE or DECODE statement must be a simple integer variable or constant.

120 S VARIABLE USED BUT NOT PREVIOUSLY DEFINED.

The variable in question has been used on the right side of an expression before it was used on the left. This error is occasionally issued in extraneous cases in CALL statements.

121 F DIAGNOSTIC TABLE OVERFLOW.

More than 8 errors occurred on this statement. The extra error messages cannot be printed.

122 F ILLEGAL FORM FOR UNIT NUMBER IN DEFINE FILE.

The unit number must be a simple integer variable or constant.

123 F MISSING "(" IN DEFINE FILE.

The left parenthesis in the DEFINE FILE statement was omitted.

124 F ILLEGAL RECORD COUNT (M) IN DEFINE FILE.

The record count designator in the DEFINE FILE statement has been incorrectly stated or omitted.

125 F ILLEGAL RECORD LENGTH (L) IN DEFINE FILE.

The record length designator in the DEFINE FILE statement has been incorrectly stated or omitted.

126 F DEFINE FILE ONLY ALLOWS UNFORMATTED (U) MODE.

It is illegal to specify any other mode other than U (unformatted mode).

127 F DEFINE FILE ASSOCIATED VARIABLE NOT A SIMPLE INTEGER.

The associated variable in a DEFINE FILE statement must be a simple integer; it must not be a constant.

128 W MISSING ")" IN DEFINE FILE.

The right parenthesis in the DEFINE FILE statement was omitted.

129 F EXPRESSION STACK OVERFLOW.

The arithmetic expression specified overflowed the stack space reserved for evaluation. Break the expression into smaller components.

130 W ILLEGAL FORM FOR OCTAL CONSTANT.

An octal constant must consist of a " followed by one to six octal digits, the value may not exceed 177777, the digits 8 and 9 may not appear in the constant.

131 W OCTAL CONSTANT TOO LARGE.

An octal constant was specified which was larger than 177777.

132 W OCTAL CONSTANT MUST HAVE AT LEAST ONE CHARACTER.

An octal constant consists of a " followed by at least one octal digit.

133 W IMPROPER FORM FOR REAL CONSTANT

A real constant was incorrectly formed. Possibly only the "." was specified.

134 I NO EXECUTABLE STATEMENTS IN A MAIN PROGRAM.

The main program has no executable statements, thus is not very meaningful.

135 W MISSING COMMA

A declarative list was specified which is missing at least one comma.

136 W REDUNDANT COMMA

An extra comma was found in a declarative list. This error occasionally occurs in combination with error 135 when illegal forms exist in the list and the Compiler is unable to recognize them.

137 W IMPLICIT STATEMENT APPEARS AFTER STATEMENT IT AFFECTS.

If an IMPLICIT statement appears after a statement it affects, the implicit typing may be done in an incomplete fashion. In some cases, the usage of the variables affected will be changed but the data attributes and array information will not.

138 W LOGICAL OP. MEANINGFUL ONLY ON BYTE, LOGICAL, OR INTEGER.

Logical operations cannot be done in a meaningful fashion on real, double precision, or complex data items.

139 F EQUIVALENCE GROUP TOO LARGE FOR ADDRESS SPACE

A group of items have been combined by an EQUIVALENCE statement so that the resulting total size of the items equivalenced exceeds 32K words. Correct by reducing the size of arrays or reorganizing the equivalence relationships.

140 W ATTEMPTED EQUIVALENCE OF MISALIGNED BYTE ITEM

EQUIVALENCE can only be used with quantities which are aligned on word boundaries. In a LOGICAL*1 (BYTE) array, the elements having odd subscripts are aligned on word boundaries. Correct by referencing only bytes having odd subscripts. For example:

LOGICAL*1 A,B(9)

correct: EQUIVALENCE(A,B(3))

incorrect: EQUIVALENCE(A,B(4))

E.2 FORTRAN COMPILER ASSEMBLY PHASE ERRORS

There are three Compiler assembly phase errors. They are:

- P - Phase error, there are two or more statements having the same statement number
- U - Undefined statement number, reference to a non-existent statement number
- M - Multiply-defined symbol, reference was made to a multiply-defined statement number. M errors are always associated with P errors and disappear when the condition causing the P error(s) is corrected.

Assembly phase error messages are printed following the list of block names in the block summary. The entire line of assembly code generated by the line containing the error is printed, preceded by the error code. FORTRAN statement numbers are shown preceded by a . character and may be followed by a : character.

A short example program is shown below along with the listing of error messages and block summary.

FORTRAN VVVV44

15:01:59

31-JUL-72

PAGE 1

```
0001      GO TO 25
0002      IF (X-1) 31, 32, 33
0003      STOP
0004      END
```

	BLOCK	LENGTH
	MAIN	40 (000120)*
U 000024	0000000	.25
U 000044	0000000	.31
M 000046	0000021	.32
U 000004	0000000	.33
P 000002	0000000.321	000000000

```
**COMPILER ***** CORE**
      PHASE      USED  FREE
DECLARATIVES  00445 17398
EXECUTABLES   00445 17398
ASSEMBLY      00897 19864
```

E.3 FORTRAN OTS ERROR DIAGNOSTICS

Each run-time error message issued by the FORTRAN OTS is of the form:

FORT $\emptyset\emptyset$ Cnnn

where C is the error class number, and nnn is the error number within the class.

Version 2 $\emptyset$ A FORTRAN OTS has 9 error classes:

<u>Error Class</u>	<u>General Error Area</u>
$\emptyset$	Very severe errors. Impossible to continue execution or to print standard error message and traceback information. The DOS Monitor diagnostic F $\emptyset$ 3 $\emptyset$ $\emptyset\emptyset\emptyset$ nnn is issued for error nnn of class $\emptyset$
1	Physical I/O errors: parity, checksum, end-of-file, etc.
2	Format specification syntax error
3	Arithmetic overflow or division by zero
4	Argument errors in function or subroutine calls
5	Arithmetic underflow
6	Conversion error for format controlled I/O
7	Subscript errors
8	Errors resulting from improper linking of OTS routines.

Some diagnostic numbers are not assigned to a specific error condition. The message

FORT $\emptyset\emptyset$ Cnn SYSTEM ERROR NO DIAGNOSTIC MESSAGE ASSIGNED

appears for these diagnostic numbers. These diagnostics should never occur during execution of a FORTRAN program.

As discussed in the description of the SETERR subroutine (Section 6.5.2), each error class has a maximum occurrence count

and a default system interpretation. The user can change the error-handling procedure through SETERR calls.

Class 0 Errors

FORT000000 INVALID CALL TO ERROR

FORTTRAN system error; an illegal argument was passed to the OTS error routine.

FORT000001 NO SPACE TO DO I/O

An attempt to open a file failed because there was not enough free memory. The program memory requirements must be reduced (see Section 11.3).

FORT000002 SUBROUTINE DIRECTLY(INDIRECTLY) REFERENCES ITSELF

A recursive subroutine call was attempted. This error may also result if the rules on overlay transfer paths (see Section 7.5.1) are violated.

FORT000003 ILLEGAL FLOATING POINT INSTRUCTION

The PDP-11/45 Floating Point Processor attempted to execute an illegal floating point instruction.

Class 1 Errors

Class 1 errors result from physical I/O errors and illegal operations on files. As described in Section 9.1.4, the END= and ERR= options in I/O statements can be used to transfer control on Class 1 errors.

FORT001000 SYSTEM ERROR NO DIAGNOSTIC MESSAGE ASSIGNED

This message should never occur.

FORT001001 DEVICE PARITY ERROR

A parity error was detected during the I/O transfer. Either the file is bad or the I/O device is malfunctioning.

FORT001002 CHECKSUM/PARITY ERROR - END OF DATA ERROR (RANDOM)

- a. A checksum or parity error occurred during unformatted I/O.
- b. A random-access I/O statement attempted to read or write beyond the physical end of the file.

FORT001003 UNDIAGNOSABLE I/O ERROR

The DOS Monitor returned an error indication during a read or write operation. The condition was not as described under FORT001001 or FORT001002.

FORT001004 END OF FILE OR END OF MEDIUM

A formatted or unformatted I/O statement has attempted to read or write beyond the physical end of the file. END= may be used to transfer control on this end of file condition.

FORT001005 UNABLE TO ALLOCATE CONTIGUOUS FILE

An attempt to allocate a contiguous file as requested by SETFIL or DEFINE FILE has failed because there is not a large enough segment of physically contiguous blocks on the device. Fragmentation of available space may be reduced by using PIP (/PK switch).

FORT001006 DEFINE FILE NOT DONE (RANDOM ACCESS)
FORT001007 DEFINE FILE DONE (NOT RANDOM ACCESS)

A DEFINE FILE statement must be executed for each random-access file, but not for any formatted or unformatted files.

FORT001008 INVALID PROTECTION FOR FILE ACCESS

The protection code for the specified file does not permit access of the type attempted by the program. See the description of the /PR switch in the PIP manual for a discussion of file protection codes.

FORT001009 FILE DOES NOT EXIST / OR IS ALREADY OPEN

On input, a READ statement attempted to open for input a non-existent file. On output, a WRITE statement attempted to create and open a linked file; a file of that name already exists in the user file directory.

FORT001010 UNABLE TO OPEN FILE

An attempt to open a file failed, but not for any of the more specific reasons described under FORT001005 through FORT001009.

FORT001011 WRONG MODE FOR FILE ACCESS

The program is attempting I/O on a file which does not support that type of I/O, e.g:
random-access I/O on a DOS linked file
unformatted I/O on a formatted ASCII file
formatted I/O on an unformatted file

FORT001012 INVALID DEVICE NUMBER

Unit number specified in I/O statement is not a legal FORTRAN logical unit number as specified in the FORTRAN device table (see Sections 10.4 and 10.5).

FORT001013 INVALID RECORD NUMBER (RANDOM ACCESS)

Record number specified in random-access I/O statement is less than 1 or greater than the maximum specified in the DEFINE FILE statement.

Class 2 Errors

FORT002000 CANNOT DO CONVERSION WITH FORMAT SPECIFIED

Execution of the format specification, including all explicit and implied repetitions, has not converted all items of the I/O list.

FORT002001 PARENTHESES NESTING TOO DEEP (>2) IN FORMAT

PDP-11 FORTRAN permits a maximum parentheses nesting depth of 2 within a format specification (not counting the outermost level).

legal: FORMAT (F10.2, 2(I3, F10.2))

legal: FORMAT (F10.2, 2(I3, 3(I3, I2)))

illegal: FORMAT (F10.2, 2(I3, 3(I3, 4(I2))))

FORT002002 SYNTAX ERROR IN FORMAT

A syntactically incorrect format specification has occurred. PDP-11 FORTRAN does minimal syntax checking of format specifications during compilation.

FORT002003 REFERENCE OUTSIDE OF RECORD BOUNDARIES

- a. An I/O list has requested transmission of more items than exist in an unformatted or direct-access record.
- b. A format specification has attempted to read or write more than 133 characters in a single ASCII record.
- c. A format specification has attempted to process more characters than specified in the buffer size parameter for ENCODE/DECODE.

Class 3 Errors

All errors in class 3 are arithmetic overflow errors. In each case, the computed result was outside the range of the representation used for that data type on the PDP-11. Arithmetic ranges are specified in Sections 2.2.1 through 2.2.8. Some of these errors (e.g. division by zero) are undefined operations.

FORT003000 SYSTEM ERROR NO DIAGNOSTIC MESSAGE ASSIGNED

FORT003001 EXPONENT OVERFLOW IN DOUBLE PRECISION ADDITION

FORT003002 EXPONENT OVERFLOW IN REAL ADDITION

FORT003003 DOUBLE PRECISION DIVISION BY ZERO

FORT003004 EXPONENT OVERFLOW IN DOUBLE PRECISION DIVISION

FORT003005 INTEGER DIVISION BY ZERO

FORT003006 EXPONENT OVERFLOW IN REAL DIVISION

FORT003007 COMPLEX DIVISION BY ZERO

FORT003008 REAL DIVISION BY ZERO

FORT003009 EXPONENT OVERFLOW IN COMPLEX MULTIPLICATION

FORT003010 EXPONENT OVERFLOW IN DOUBLE PRECISION MULT.

FORT003011 EXPONENT OVERFLOW DURING NEGATION
 FORT003012 EXPONENT OVERFLOW IN REAL MULTIPLICATION
 FORT003013 INTEGER OVERFLOW ON ADDITION
 FORT003014 PRODUCT OUTSIDE OF RANGE ON INTEGER MULT.
 FORT003015 INTEGER BASE = 0, INTEGER EXPONENT <= 0
 FORT003016 DOUBLE BASE = 0, INTEGER EXPONENT <= 0
 FORT003017 DOUBLE BASE = 0, DOUBLE EXPONENT <> 0
 FORT003018 DOUBLE BASE < 0, DOUBLE EXPONENT <> 0
 FORT003019 REAL BASE = 0, REAL EXPONENT <> 0
 FORT003020 REAL BASE < 0, REAL EXPONENT <> 0
 FORT003021 REAL BASE = 0, INTEGER EXPONENT <= 0
 FORT003022 REAL OUTSIDE RANGE ON REAL TO INTEGER CONVERSION
 FORT003023 EXPONENT OVERFLOW ON DOUBLE TO REAL CONVERSION
 FORT003024 FLOATING POINT EXPONENT OVERFLOW
 FORT003025 FLOATING POINT DIVISION BY ZERO
 FORT003026 INTEGER OVERFLOW ON DO-LOOP VARIABLE
 FORT003027 COMPLEX BASE = 0, INTEGER EXPONENT <= 0

Class 4 Errors

Class 4 errors are issued by library subroutines and functions.
 Each message identifies the routine issuing the message and the cause.
 See Chapter 6 for descriptions of these routines.

FORT004000 SYSTEM ERROR NO DIAGNOSTIC MESSAGE ASSIGNED
 FORT004001 SYSTEM ERROR NO DIAGNOSTIC MESSAGE ASSIGNED
 FORT004002 DEXP CALLED WITH EXPONENT GREATER THAN 87.
 FORT004003 DLOG ARGUMENT LESS THAN OR EQUAL TO ZERO
 FORT004004 DSQRT ARGUMENT LESS THAN ZERO
 FORT004005 EXP CALLED WITH EXPONENT GREATER THAN 87.
 FORT004006 ARGUMENTS OUT OF RANGE FOR "TIME" CONVERSION
 FORT004007 IABS ABS(X) GREATER THAN 2**14 - 1
 FORT004008 IDIM RESULT OUTSIDE OF RANGE -2**15-1 TO 2**15-1
 FORT004009 ISIGN RESULT GREATER THAN 2**15 - 1
 FORT004010 ALOG ARGUMENT LESS THAN OR EQUAL TO ZERO
 FORT004011 SQRT ARGUMENT LESS THAN ZERO
 FORT004012 SNGL EXPONENT OVERFLOW ON ROUND
 FORT004013 RANDU/RAN WRONG NUMBER OF ARGUMENTS
 FORT004014 PDUMP WRONG NUMBER OF ARGUMENTS

FORT004015 INVALID ARGUMENT TO "SETFIL"

Issued by SETFIL (see Section 6.5.3) for any of the following reasons:

- a) unit number 50, or greater than maximum unit number permitted by \$DEVTB (see Section 10.5).
- b) Allocate file argument, CS, is not 1 or 2.
- c) Allocation of a contiguous file for non-random I/O was requested, but additional required parameters were not supplied.

FORT004016 FILE ALREADY OPEN ON UNIT - "SETFIL" IGNORED

The logical unit specified to SETFIL has been opened and not closed. ENDFILE can be used to close a file and make the logical unit available for other uses.

Class 5 Errors

All errors in this class are arithmetic exponent underflow errors. An attempt was made to compute a non-zero number smaller than the smallest representable number on the PDP-11. The result is set to 0.0.

FORT005000 CSQRT UNDERFLOW
FORT005001 EXPONENT UNDERFLOW ON DOUBLE PRECISION ADDITION
FORT005002 EXPONENT UNDERFLOW ON REAL ADDITION
FORT005003 EXPONENT UNDERFLOW ON REAL DIVISION
FORT005004 DEXP CALLED WITH EXPONENT LESS THAN -88.7
FORT005005 EXP CALLED WITH EXPONENT LESS THAN -88.7
FORT005006 EXPONENT UNDERFLOW ON DOUBLE MULTIPLICATION
FORT005007 EXPONENT UNDERFLOW ON REAL MULTIPLICATION
FORT005008 EXPONENT UNDERFLOW ON DOUBLE PRECISION DIVISION
FORT005009 FLOATING POINT EXPONENT UNDERFLOW

Class 6 Errors

FORT006000 CONVERSION ERROR

An error occurred during a format conversion.

On output conversions, the error may result from:

- a) a value too large to fit in specified field, or
- b) an L conversion done on a value which is neither .TRUE nor .FALSE.

The output field is filled with asterisks (*)

On input conversions, the error may result from:

- a) a value too large to be represented in specified PDP-11 data type, or
- b) illegal characters in the input field.

Class 7 Errors

FORT007000 SUBSCRIPT LESS THAN OR EQUAL TO ZERO

An array subscript is ≤ 0 . This is never legal in FORTRAN IV. A lower limit check is made on every subscript reference in PDP-11 FORTRAN.

FORT007001 SUBSCRIPT GREATER THAN DIMENSIONED

An array subscript is greater than the maximum value specified in the DIMENSION statement for the array. The upper limit check is optional and is made only if the /CK switch is set prior to compilation of the program or subroutine (see Section 12.2).

FORT007002 VALUE OUT OF BOUNDS (COMPUTED OR ASSIGNED GO TO)

This error results when:

- a) The index value is greater than the length of the label list in a computed GO TO.
- b) The value of the variable is not among those in the optional label list for an assigned GO TO.

In either case, the GO TO is not executed and program execution continues at the next statement in the FORTRAN program.

FORT007003 FLOATING POINT UNDEFINED VARIABLE

A PDP-11/45 floating-point undefined variable interrupt has occurred. PDP-11 FORTRAN does not enable this interrupt. This message should not occur.

Class 8 Errors

The FORTRAN OTS I/O packages contain a number of interdependent modules. If a FORTRAN program does not use certain I/O routines (e.g. unformatted I/O), short dummy routines are substituted for the regular I/O routine during the linking process. If one of these dummy I/O routines is called, a Class 8 message results and program execution is terminated.

Class 8 errors ordinarily result from improper structuring of an overlay system (see Chapter 7) or the use of a format specification stored in an array or read at execution time. Forcing the loading of the required I/O and conversion routines will correct the problem.

FORT008000 LINKAGE ERROR (MISSING FORMAT CONVERSION ROUTINE)

A numeric format conversion routine required by the program was not included. Including a dummy FORMAT statement in the (resident section) program will force loading of conversion routines. (See Section 7.3.)

FORT008001 LINKAGE ERROR (UNFORMATTED I/O)

An overlay attempted to do unformatted I/O, but the required control routines were not included in the resident section (see Section 7.3).

FORT008002 LINKAGE ERROR (DIRECT-ACCESS I/O)

An overlay attempted to do direct-access I/O, but the required control routines were not included in the resident section (see Section 7.3).

APPENDIX F

FORTTRAN DEVICE TABLE LISTING

(\$DEVTB)

```

.TITLE $DEV879
.GLOBAL $DEVTB
.CSECT
;
; $DEVTB V009A
;
; COPYRIGHT 1971,1972 DIGITAL EQUIPMENT CORPORATION, MAYNARD,MASS
;
; THESE ARE THE FORTTRAN DEVICE TABLE ENTRIES
; WITH THE DEVICE TABLE HEADER AND ENTRY VECTOR
;
;
; .IFNDF RSX
; .WORD DEV43 ;ADDR OF ENTRY FOR ERR LOG DEVICE
; ;ERROR LOG DEVICE IS -3(DEV43)
; .WORD DEVERR ;ADDR OF ENTRY FOR ERR MSG FILE
; .ENDC
$DEVTB: .WORD 9. ;NUMBER OF ENTRIES IN ENTRY VECTOR
; .IFNDF RSX
; .WORD -3 ;DEVICE NUM OF ERROR LOGGING DEVICE
; .ENDC
; .IFDF RSX
; .WORD 5. ;DEVICE NUM OF ERROR LOGGING DEVICE
; .ENDC
;
; THE DEVICE TABLE ENTRY VECTOR
;
; .WORD DEV1 ;ADDR OF DEVICE 1 ENTRY
; .WORD DEV2 ;ADDR OF DEVICE 2 ENTRY
; .WORD DEV3 ;ADDR OF DEVICE 3 ENTRY
; .WORD DEV4 ;ADDR OF DEVICE 4 ENTRY
; .WORD DEV5 ;ADDR OF DEVICE 5 ENTRY
; .WORD DEV6 ;ADDR OF DEVICE 6 ENTRY
; .WORD DEV7 ;ADDR OF DEVICE 7 ENTRY
; .WORD DEV8 ;ADDR OF DEVICE 8 ENTRY
;
;
; ENTRY 1 OF DEVICE TABLE
;
; DEV1: .WORD 0 ;LINK BLOCK PTR
; .IFNDF RSX
; .RAD50 /SY / ;PHYSICAL DEVICE NAME DEFAULT
; .ENDC
; .IFDF RSX
; .RAD50 /43/
; .ENDC
; .BYTE 0 ;HOW OPEN SWITCH
; .BYTE 1 ;UNIT NUM DEFAULT
; .RAD50 /FOR/ ;DEFAULT FILE NAME
; .RAD50 /001/

```

.RAD50	/DAT/	;DEFAULT EXTENSION
.BYTE	233	;NO AUTO DEL, GROUP & OTHERS READ/RUN ONLY
.BYTE	0	;DEVICE STATUS SWITCH
.BYTE	0	;MODE OF I/O - FUNCN WORD (RANDOM)
.BYTE	0	;STATUS OF I/O
.WORD	0	;RECORD COUNT - BLOCK NUM (RANDOM)
.WORD	0	;BUFF ADDR (RANDOM)
.WORD	0	;BUF LEN (RANDOM)
.WORD	0	;ASSOCIATED VAR ADDR (FROM DEFINE FILE)
.WORD	0	;NUM RECORDS IN FILE (FROM DEFINE FILE)
.WORD	0	;RECORD LENGTH (FROM DEFINE FILE)
.WORD	0	;USER ID CODE
.WORD	0	;ERROR VAR ADDR (FROM SETFIL)

;
;
;

;ENTRY 2 OF DEVICE TABLE

DEV2: .WORD 0
 .IFNDF 00X
 .RAD50 /SY /
 .ENDC
 .IFDF 00X
 .RAD50 /TT/
 .ENDC
 .BYTE 0,0
 .RAD50 /FOR/
 .RAD50 /002/
 .RAD50 /DAT/
 .BYTE 233,0,0,0
 .WORD 0,0,0,0,0,0,0,0

;
;
;

;ENTRY 3 OF DEVICE TABLE

DEV3: .WORD 0
 .IFNDF 00X
 .RAD50 /SY /
 .ENDC
 .IFDF 00X
 .RAD50 /00/
 .ENDC
 .BYTE 0,0
 .RAD50 /FOR/
 .RAD50 /003/
 .RAD50 /DAT/
 .BYTE 233,0,0,0
 .WORD 0,0,0,0,0,0,0,0

;
;
;

;ENTRY 4 OF DEVICE TABLE

DEV4: .WORD 0
 .RAD50 /00 /
 .BYTE 0,0
 .RAD50 /FOR/
 .RAD50 /004/
 .RAD50 /DAT/
 .BYTE 233,0,0,0
 .WORD 0,0,0,0,0,0,0,0

```

;
;
;ENTRY 5 OF DEVICE TABLE
;
;
DEV5:  .WORD    3
        .RAD50  /LP /
        .BYTE   3,3
        .RAD50  /FDR/
        .RAD50  /005/
        .RAD50  /DAT/
        .BYTE   233,0,0,0
        .WORD   0,0,0,0,0,0,0,0

```

```

;
;
;ENTRY 6 OF DEVICE TABLE
;
;
DEV6:  .WORD    3
        .RAD50  /KB /
        .BYTE   3,0
        .RAD50  /FDR/
        .RAD50  /006/
        .RAD50  /DAT/
        .BYTE   233,0,0,0
        .WORD   1,0,0,0,0,0,0,0

```

```

;
;
;ENTRY 7 OF DEVICE TABLE
;
;
DEV7:  .WORD    3
        .IFNDF  RSX
        .RAD50  /SY /
        .ENDC
        .IFDF   RSX
        .RAD50  /JD/
        .ENDC
        .BYTE   3,0
        .RAD50  /FDR/
        .RAD50  /007/
        .RAD50  /DAT/
        .BYTE   233,0,0,0
        .WORD   1,0,0,0,0,0,0,0

```

```

;
;
;ENTRY 8 OF DEVICE TABLE
;
;
DEV8:  .WORD    3
        .IFNDF  RSX
        .RAD50  /BI /
        .ENDC
        .IFDF   RSX
        .RAD50  /AD/
        .ENDC
        .BYTE   3,3
        .RAD50  /FDR/
        .RAD50  /008/
        .RAD50  /DAT/
        .BYTE   233,0,0,0
        .WORD   0,0,0,0,0,0,0,0
        .IFNDF  RSX

```

```

;
;ENTRY = 3 OF DEVICE TABLE(ERROR LOGGING DEVICE)
;SPECIAL ENTRY USED AS DJS-11 ERROR LOGGING DEVICE
;LOGICAL UNIT NUMBER = -3. LOGICAL DEVICE NAME = CM0

```

```

;
DEV43: .WORD 3 ;LINK BLOCK PTR
       .RAD50 /KR / ;DEFAULT PHYSICAL DEVICE NAME
       .BYTE 3,3
       .RAD50 /FDR/ ;DEFAULT FILE NAME
       .RAD50 /CM0/ ;FORCM0.DAT
       .RAD50 /DAT/
       .BYTE 233,0,0,0 ;
       .WORD 3,3,0,0,0,3,3,0 ;

```

```

;
;SPECIAL ENTRY FOR ERROR PROCESSORS MSG FILE

```

```

;
DEVERR: .WORD 3 ;LINK BLOCK ERR RTN ADDR
        .WORD 3 ;LINK PTR
        .RAD50 /ERR/ ;LOG DATA SET NAME
        .BYTE 1 ;PHYSICAL DS NAME FOLLOWS
        .BYTE 3 ;UNIT NUM
        .RAD50 /SY / ;PHYSICAL DEVICE NAME
        ;DEFAULT TO SYSTEM DEVICE
        .WORD 3 ;FILE BLOCK ERROR RETURN ADDR
        .BYTE 4 ;HOW TO OPEN (OPENI)
        .BYTE 3 ;ERROR RTN CODE
        .RAD50 /FDR/ ;FILE NAME
        .RAD50 /RJN/
        .RAD50 /JGN/
        .BYTE 1 ;USER ID CODE
        .BYTE 1
        .BYTE 322,0 ;ALLOW ONLY INPUT ACCESS
        .WORD 4 ;FUNCTION WORD (READ)
        .WORD 3 ;BLOCK NUM
        .WORD 3 ;BLOCK ADDR
        .WORD 3 ;BLOCK LENGTH

```

```

;
;
.ENDC
.END

```

INDEX

- Adjustable arrays, 6-8, 6-6, 5-6
- A format conversions, 8-11
- ANSI, 1-1
 - additions to, 1-1
- Apostrophe, 1-1
- Arithmetic assignment statement, 3-1, 3-3
- Arithmetic statement functions, 6-2
- Array format specifications, 8-16 8-17
- Array subscripts, 1-1, 2-9
- Array variables, 2-8
 - adjustable, 5-6, 6-6, 6-8
 - dimensioning, 2-9, 5-5
 - equivalencing, 5-8, 5-12
 - storage, 2-9, 5-6
 - type 5-6
- ASCII code, 2-1, A-1
- Assembly language
 - library routines, 12-6
 - overlays, 7-13
 - subprograms, 6-13
- ASSIGN, 3-2
- ASSIGN Monitor command, 7-16
 - 10-9, 10-10, 10-11
- Associated variable, 9-11
- /AS switch option, 12-6

- BACKSPACE, 9-14
- BEGIN Monitor command, 7-16
- Binary I/O, 9-9, 9-10, 10-6, 10-7
- BLOCK DATA, 6-15, 7-32
- Block summary, 12-8
- BYTE Variables, 2-8, 5-13
 - format, B-3
 - see also LOGICAL*1

- CALL, 6-12
- Calling an overlay file, 7-6
- Card codes, A-3
- Carriage control, 8-14, 10-5, 10-6
- Character set, 2-1, A-1
- Character string arguments to system subroutines, 6-26, 7-6
- Character string format
 - specifications, 8-16, 8-17
- /CK switch option, 5-6, 12-7
- Coding form, 1-4
- Comment lines, 1-6
- COMMON, 5-8, 2-9
 - associated variable, 9-11
 - blank common, 5-8
 - dummy arguments, 6-7, 6-8
- COMMON (Cont.)
 - EQUIVALENCE interaction, 5-13
 - overlay systems, 5-10, 7-1, 7-14
 - size, 5-10
 - storage allocation, 5-10
 - used with independently linked programs, 7-32
 - used with subprograms, 6-1
- Compilation procedures, 12-3
- memory requirements, 12-10
- Compile-load-go, 12-5, 12-7
- Complex constants, 2-5
 - expressions, 2-14
 - format, B-3
- Complex I/O formats, 8-9
- Constants, 2-1
- CONTINUE statement, 4-7
- CONTINUE Monitor command, 4-8
- /CO switch option, 12-7, 12-10
- Contiguous files, 10-3, 10-5, 10-7
- Continuation lines, 1-5, 1-6
- Conversion rules, 3-3, 2-12

- DATA, 5-15, 1-1, 6-7
- Data types, 1-1, B-1
- DATE subroutine, 6-22
- DECØ26 punch, A-3
- DECØ29 punch, A-3
- Declarations
 - see specification statements
- DECODE, 9-15
- Device data block, 11-6
- Device driver sizes, 11-6
- DEFINE FILE, 9-10, 7-16, 10-7
 - used in example, 10-13
- Descriptor block, 12-8
- Devices, 9-3, 10-4, 10-9, 10-10, 10-11
 - effect on memory requirements, 12-10
 - specification codes, 12-2
- Device table, 10-2, 10-3, 10-11, 10-12, 10-13
 - listing, F-1
- \$DEVTB, 10-2, 10-3, 10-11, 10-12, 10-13
 - listing, F-1
- DDB, see device data block
- D format conversions, 8-6
- DIMENSION, 5-5, 2-9, 5-10
- Direct Access I/O, 9-10
 - DEFINE FILE, 9-10
 - FIND, 9-12
 - READ, 9-12
 - WRITE, 9-13

DO, 4-4
 DOS Monitor, 1-2,
 error messages, 7-12, 7-16
 Double-precision, 2-14
 constants, 2-3
 format, B-2
 Dummy arguments, 5-15, 6-5, 6-6,
 6-8, 6-9, 6-11, 6-12
 DVBØ9.PAL file, 10-11

Edit-11, 1-5
 E format conversions, 8-5
 ENCODE, 9-15
 END, 1-6, 4-9, 6-8, 6-11, 6-13
 END=n, 1-1, 9-14, 10-2, 10-9
 END FILE, 9-15, 11-5, 11-6
 EQUIVALENCE, 5-10
 array variables, 5-12
 BYTE or LOGICAL*1, 5-13
 COMMON interaction, 5-13
 dummy arguments, 6-7
 /ER switch option, 12-7
 ERR=n, 1-1, 9-14, 10-2, 10-9
 Error classes, 6-18, 11-10, E-16
 Error handling, E-1
 assembly phase errors, E-15
 compiler errors, E-1
 I/O errors, 10-8
 OTS error processing, 11-9, E-16
 Error logging device, 9-3
 Executable statements, 1-3, 1-6,
 5-4, 5-5, 5-9, 5-14, 6-2, 8-2,
 9-10
 EXIT subroutine, 4-9, 6-22
 Expressions, 2-9
 arithmetic, 2-10
 logical, 2-13, 2-16
 mixed mode, 2-11
 parentheses, 2-11, 2-14, 2-15
 relational, 2-14, 2-16
 EXTERNAL, 5-14

F format conversions, 8-4
 FIB, see
 file information block
 File information block, 11-6
 Filename specifications, 12-2
 assumed/default extensions,
 12-3
 Files
 contiguous, 10-3, 10-5, 10-7
 linked, 10-3, 10-5
 FIND, 9-12
 Floating point, see real or
 double precision
 FORMAT, 8-1
 alphanumeric data, 8-12
 carriage control, 8-14

FORMAT (Cont.)
 ENCODE/DECODE, 9-15
 field repetition, 8-15
 field specifications, 8-2
 Format specifications, 1-1
 Formatted I/O, 9-6, 10-5
 PRINT, 9-8
 READ, 9-6
 WRITE, 9-7
 Four-word floating point, see
 double precision
 FUNCTION, 6-8
 EXTERNAL, 5-14, 6-1
 name, 6-9
 Function definitions
 see FUNCTION
 statement functions, 1-6, 6-2
 Functions, see library functions

G format conversions, 8-8
 /GO switch option, 12-7
 GOTO
 assigned, 1-1, 4-2
 computed, 4-1
 unconditional, 4-1

Hardware environment, 1-2
 Hexadecimal constants, 1-1, 2-4
 Hollerith constants, 1-1, 2-6
 format, B-3
 I/O conversion, 8-12

IF
 arithmetic, 4-3
 logical, 4-4
 I format conversion, 8-3
 IMPLICIT, 1-1, 1-6, 5-1
 Integers, 2-14
 constants, 2-2
 format, B-1
 variables, 2-8
 I/O lists, 9-4
 I/O
 single buffered, 10-2, 10-11

KILL Monitor command, 4-8

L format conversions, 8-10
 Library, see OTS
 Library functions, 5-3, 6-2, 6-3,
 6-4, 12-6
 algorithms, D-1
 LINK subroutine, 7-4
 loads overlay file, 7-5
 moves stack, 7-5
 RETURN entry point, 7-5

LINK subroutine (Cont.)
 saves registers, 7-5
 traceback, 7-4
 Link-11, 7-1, 7-9, 12-4
 command string format, 7-9
 operations performed, 7-10
 relinking, 7-11
 Linking procedures, 12-3
 Listing format, 12-8
 Lists, I/O; see I/O lists
 /LI switch option, 12-6, 12-8
 Logical
 constants, 2-13
 expressions, 2-13, 2-16
 Logical assignment
 statement, 3-2
 Logical constants, 2-6
 format, B-4
 LOGICAL*1 variables, 2-8, 5-13
 format, B-3
 Logical unit numbers, 9-3, 10-2,
 10-9
 Loops, 4-4

MACRO-11 Assembler, 1-2
 linkages, 6-13
 parameters, 6-13
 subprograms, 6-13
 variable number of arguments,
 6-14
 Memory map
 compile time, 12-10
 runtime, 11-5
 Mixed mode expressions, 1-1, 2-11
 2-12
 Multiple record transmission,
 8-15

Non-executable statements, 1-3,
 5-4, 5-5, 5-9, 5-14, 6-2, 8-1,
 9-10

Object Time System, see OTS
 Octal constants, 1-1, 2-4, 4-8
 ODT, 11-7
 O format conversions, 8-4
 /ON switch option, 12-7
 Operating procedures, 12-1
 compilation/linking, 12-3
 compile/load/go, 12-5, 12-7
 overlays, 7-9
 switch options, 12-6
 Operators
 arithmetic, 2-10, 2-16
 logical, 2-15, 2-16
 relational, 2-14, 2-16

OTS, 10-1
 carriage control, 10-6
 error processing, 11-9
 function algorithms, D-1
 library usage, 12-6
 segment control word, 10-6
 threaded code, 11-1
 Output, see listing format
 Overlay facility, 7-1, 7-17
 assembly language overlays, 7-13
 calling overlay, 7-6
 communication, 7-2
 control paths, 7-7
 error handling, 7-11
 format conversions, 7-2
 I/O routines, 7-2
 operating procedures, 7-9
 programming example, 7-17
 search for files, 7-8

PAL-11R Assembler, 1-2
 Parentheses, 2-11, 2-14, 2-15
 Parity ASCII, A-1, A-3
 PAUSE, 4-8
 PDUMP subroutine, 6-16, 11-7
 Polish mode
 breakpoints in, 11-7
 entry, 11-3
 exit, 11-4
 Polish notation, 11-1
 PRINT, 9-8
 Procedures, 6-1
 Program unit, 1-6
 Punched card codes, A-3

Random access I/O, 1-1
 RANDU subroutine, 6-21
 READ
 direct access, 9-12
 formatted, 9-6
 operation, 10-1
 transfer of control, 9-13
 unformatted, 9-9
 Real constants, 2-3, 2-14, B-1, B-2
 Record, 8-1, 8-2, 9-3, 10-1
 direct access processing, 10-7
 field specifications, 8-2, 8-15
 formatted, 10-5
 logical record segment format,
 10-7
 multiple, 8-15
 spacing control, 8-13
 unformatted, 10-6

Registers
 general, 7-1, 7-5, 7-14
 R4, 11-2

- Relational expressions, 2-14
- RETURN, 4-9, 6-8, 6-11, 6-12
 - CALL RETURN, 7-6, 7-7
- REWIND, 9-14
- RUN subroutine (entry point to LINK subroutine), 7-32
- Run-time
 - memory organization, 11-5
 - overlay supervisor, 7-1, 7-4
 - see LINK subroutine

- Scalar variable, 2-8
- Scale factors, 8-9
- Segment control word, 10-6
- SETERR subroutine, 6-18
- SETFIL subroutine, 6-19, 7-16, 10-3, 10-9, 10-10
 - used in example, 10-13
- Software environment, 1-2
- Specification statements, 1-6, 5-1, 6-2
- SSWTCH subroutine, 6-25
- Stack, 7-7, 7-13, 11-5
- Statement field, 1-5
- Statement number field, 1-5
- Statement order, 1-6
- STOP, 4-8
- Subprogram calls,
 - parameters passed, 6-1, 6-5, 6-6
 - recursiveness, 6-1
- Subprograms
 - assembly language, 6-13
 - EXTERNAL, 5-14
 - names, 5-14
 - statements, 1-6, 6-1
 - see also FUNCTION and SUBROUTINE
- SUBROUTINE, 6-9
 - EXTERNAL, 5-14, 6-1
 - system, 6-16
- /SU switch option, 12-7
- Switch options, 12-6
- System subroutines
 - character string arguments, 6-26
 - DATE, 6-22
 - EXIT, 6-22
 - LINK, 6-26, 7-4
 - PDUMP, 6-16, 11-7
 - RANDU, 6-21
 - RUN, 6-26
 - SETERR, 6-18
 - SETFIL, 6-19, 6-26, 7-16
 - SSWTCH, 6-25
 - TIME, 6-23
 - TSTERR, 6-25

- Tab character, 1-5, 2-1
- Threaded code, 11-1, 11-7
- TIME subroutine, 6-23
- Traceback feature, 11-7
 - in overlays, 7-4
- Transfer control on I/O error
 - condition, 1-1, 9-14, 10-2, 10-9
- TSTERR subroutine, 6-25
- Two-word floating point see real constants
- Type declaration, 2-9, 5-2, 5-3, 5-6, 5-10

- Unformatted I/O, 10-6
 - READ, 9-9
 - WRITE, 9-10

- Variable read-in formats, 9-7
- Variables, 2-7

- .WAIT macro, 10-2
- Word formats, B-1
- WRITE
 - direct access, 9-13
 - formatted, 9-7
 - operation, 10-1
 - transfer of control, 9-13
 - unformatted, 9-10

- X format specification, 8-13

HOW TO OBTAIN SOFTWARE INFORMATION

Announcements for new and revised software, as well as programming notes, software problems, and documentation corrections are published by Software Information Service in the following newsletters.

Digital Software News for the PDP-8 & PDP-12
Digital Software News for the PDP-11
Digital Software News for the PDP-9/15 Family

These newsletters contain information applicable to software available from Digital's Program Library. Articles in Digital Software News update the cumulative Software Performance Summary which is contained in each basic kit of system software for new computers. To assure that the monthly Digital Software News is sent to the appropriate software contact at your installation, please check with the Software Specialist or Sales Engineer at your nearest Digital office.

Questions or problems concerning Digital's Software should be reported to the Software Specialist. In cases where no Software Specialist is available, please send a Software Performance Report form with details of the problem to:

Software Information Service
Digital Equipment Corporation
146 Main Street
Maynard, Massachusetts 01754

These forms which are provided in the software kit should be fully filled out and accompanied by teletype output as well as listings or tapes of the user program to facilitate a complete investigation. An answer will be sent to the individual and appropriate topics of general interest will be printed in the newsletter.

Orders for new and revised software and manuals, additional Software Performance Report forms, and software price lists should be directed to the nearest Digital Field office or representative. U.S.A. customers may order directly from the Program Library in Maynard. When ordering, include the code number and a brief description of the software requested.

Digital Equipment Computer Users Society (DECUS) maintains a user library and publishes a catalog of programs as well as the DECUSCOPE magazine for its members and non-members who request it. For further information please write to:

DECUS
Digital Equipment Corporation
146 Main Street
Maynard, Massachusetts 01754



1

2



3

4



PDP-11 FORTRAN IV
Programmer's Manual
DEC-11-LFIVA-A-D
August, 1972

READER'S COMMENTS

Digital Equipment Corporation maintains a continuous effort to improve the quality and usefulness of its publications. To do this effectively we need user feedback -- your critical evaluation of this manual.

Please comment on this manual's completeness, accuracy, organization, usability and readability.

Did you find errors in this manual? If so, specify by page.

How can this manual be improved?

Other comments?

Please state your position. _____ Date: _____

Name: _____ Organization: _____

Street: _____ Department: _____

City: _____ State: _____ Zip or Country _____

----- **Fold Here** -----

----- Do Not Tear - Fold Here and Staple -----

**FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.**

[REDACTED]

**Digital Equipment Corporation
Software Information Services
146 Main Street
Maynard, Massachusetts 01754**

PDP-11 FORTRAN IV
Programmer's Manual
DEC-11-LFIVA-A-D
August, 1972

READER'S COMMENTS

Digital Equipment Corporation maintains a continuous effort to improve the quality and usefulness of its publications. To do this effectively we need user feedback -- your critical evaluation of this manual.

Please comment on this manual's completeness, accuracy, organization, usability and readability.

Did you find errors in this manual? If so, specify by page.

How can this manual be improved?

Other comments?

Please state your position. _____ Date: _____

Name: _____ Organization: _____

Street: _____ Department: _____

City: _____ State: _____ Zip or Country _____

----- **Fold Here** -----

----- **Do Not Tear - Fold Here and Staple** -----

**FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.**

[REDACTED]

**Digital Equipment Corporation
Software Information Services
146 Main Street
Maynard, Massachusetts 01754**

**Digital Equipment Corporation
Software Information Services
146 Main Street
Maynard, Massachusetts 01754**